Decentralized enforcement of document lifecycle constraints

Sylvain Hallé, Raphaël Khoury, Quentin Betti, Antoine El-Hokayem, Yliès Falcone

▶ **To cite this version:**

Sylvain Hallé, Raphaël Khoury, Quentin Betti, Antoine El-Hokayem, Yliès Falcone. Decentralized enforcement of document lifecycle constraints. Information Systems, 2017, 10.1016/j.is.2017.08.002 . hal-01653879

HAL Id: hal-01653879
https://hal.science/hal-01653879

Submitted on 2 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Decentralized Enforcement of Document Lifecycle Constraints

Sylvain Hallé[1], Raphaël Khoury[1], Quentin Betti[1], Antoine El-Hokayem[2], Yliès Falcone[2]

Abstract

Artifact-centric workflows describe possible executions of a business process through constraints expressed from the point of view of the documents exchanged between principals. A sequence of manipulations is deemed valid as long as every document in the workflow follows its prescribed lifecycle at all steps of the process. So far, establishing that a given workflow complies with artifact lifecycles has mostly been done through static verification, or by assuming a centralized access to all artifacts where these constraints can be monitored and enforced. We present in this paper an alternate method of enforcing document lifecycles that requires neither static verification nor single-point access. Rather, the document itself is designed to carry fragments of its history, protected from tampering using hashing and public-key encryption. Any principal involved in the process can verify at any time that the history of a document complies with a given lifecycle. Moreover, the proposed system also enforces access permissions: not all actions are visible to all principals, and one can only modify and verify what one is allowed to observe. These concepts have been implemented in a software library called Artichoke, and empirically tested for performance and scalability.

1. Introduction

The execution of a business process is often materialized by the successive manipulation of a document passing from one agent to the next. However, the document may have constraints on the way it is modified, and by whom: we call this the lifecycle of a document. In the past decade, *artifact-centric* business processes have been suggested as a modelling paradigm where business process workflows are expressed solely in terms of document constraints: a sequence of manipulations is deemed valid as long as every document (or "artifact") in the workflow follows its own prescribed lifecycle at all steps of the process. In this context, an artifact becomes a stateful object, with a finite-state machine-like expression of its possible modifications. As we shall see in Section 2, this

[1]Laboratoire d'informatique formelle, Université du Québec à Chicoutimi, Canada
[2]Université Grenoble Alpes, Inria, LIG, Grenoble, France

paradigm can be applied to a variety of situations, ranging from medical document processing to accounting and even electronic-pass systems such as smart cards.

Central to the question of business processes execution is the concept of *compliance checking*, or the verification, through various means, that a given implementation of a business process satisfies the constraints associated with it. Transposed to artifact-centric business processes, this entails that one must provide some guarantee that the lifecycle of each artifact involved is respected at all times.

There are currently two main approaches to the enforcement of this lifecycle, which will be detailed in Section 3. A first possibility is that all the peers involved in the manipulations trust each other and assume they perform only valid manipulations of the document; this trust can be assessed through testing or static verification of the peer's implementation. Otherwise, all peers can trust a third party, through which all accesses to the document need to be done; this third-party is responsible for enforcing the document's lifecycle, and must prevent invalid modifications from taking place. The reader shall note that both scenarios require some form of external trust, which becomes an entry point for attacks. In the first scenario, a single malicious user can thwart the enforcement of the lifecycle and invalidate any guarantees the other peers can have with respect to it. In the second scenario, reliance on a third party opens the way to classical mistrust-based attacks, such as man-in-the-middle.

In this paper, we present a mechanism for the distributed enforcement of a document's lifecycle, in which every peer can individually check that the lifecycle of a document it is being passed is correctly followed. It is an extended version of a previously published work [1]. Section 4 first shows how various aspects of a document lifecycle can be expressed formally using a variety of specification languages. Our proposed system, presented in Section 5, requires neither centralized access to the document, nor trust in other peers that are allowed to manipulate it. Rather, the document itself is designed to carry fragments of its history, called a *peer-action sequence*. This sequence is protected from tampering through careful use of hashing and public-key encryption. Using this system, any peer involved in the business process can verify at any time that a document's history complies with a given lifecycle, expressed as a finite-state automaton. Moreover, the proposed system also enforces access permissions: not all actions are visible to all principals, and one can only modify and verify what one is allowed to observe.

To illustrate the concept, Section 6 revisits one of the use cases described in the beginning of the paper, and shows how lifecycle constraints can be modelled and enforced using our proposed model. Section 7 then describes an implementation of these principles in a simple command-line tool that manipulates dynamic PDF forms. Peer-action sequences are injected through a hidden field into a PDF file, and updated every time the form is modified through the tool. As a result, it is possible to retrieve the document's modification history at any moment, verify its authenticity using a public keyring, and check that it complies with a given policy.

This paper extends the original publication on multiple aspects. First, it

provides a complete formalization of all the relevant concepts, a task that could not be done before, due to lack of space. It also provides several enhancements to the original ideas, including multi-group actions, prefix deletion, and permission revocation. Finally, it describes a new software tool that implements this ideas, and which has been tested in a new set of experiments.

The paper concludes with a few discussion points. In particular, it highlights the fact that, using peer-action sequences, the compliance of a document with a given lifecycle specification can easily be checked. Taken to the extreme, lifecycle policies can even be verified without resorting to any workflow management system at all: as long as documents are properly stamped by every peer participating in the workflow, the precise way they are exchanged (e-mail, file copying, etc.) becomes irrelevant. This presents the potential of greatly simplifying the implementation of artifact-centric workflows, by dropping many assumptions that must be fulfilled by current systems.

2. Document Lifecycles

We shall first describe a number of distinct scenarios, taken from past literature, that can be modelled as sets of constraints over the lifecycle of some document. In the following, the term *document* will encompass any physical or logical entity carrying data and being passed on to undergo modifications. This can represent either a physical memory card, a paper or electronic form, or more generally, any object commonly labelled as an "artifact" in some circles.

A special case of "lifecycle" is one where conditions apply on snapshots of documents taken individually, irrespective of their relation with previous or subsequent versions of this document. For example, the lifecycle could simply express conditions on what values various elements of a document can take, and be likened to integrity constraints. However, in the following, we are more interested in lifecycles that also involve the *sequence* of states in which the document is allowed to move through, and the identity of the effectors of each modification.

2.1. Medical Document Processing

We first illustrate our approach with the following example, adapted from the literature [2, 3], of a document that registers medically-relevant information related to a given patient. A decentralized enforcement of such a medical file would offer multiple advantages, since it would facilitate collaboration between multiple health-care providers who are required to adhere to their own guidelines with respect to the confidentiality of health care information and will allow medical data to be shared safely between institutions.

Such a medical document typically includes several of pieces of information, notably:

- the patient's identifying information,
- an insurance policy number,

- a series of tests, requested by a doctor, and performed by medical professionals,
- drug prescriptions, filled by a pharmacist with the approval of the patient's insurance company.

Each of these pieces of information can be input into the document either as an atomic value or string (the insurance number and the personal information), as a list (tests), as a map (a single test, comprising a request that it be undertaken mapped to the results of the test). Access to the document is shared between a patient, a doctor and other medical staff, the insurance company and the pharmacist, each possessing distinct access control rights.

Given the sensitive nature of the information contained in such a document, it goes without saying that different principals are subject to different privileges to read, write, or modify each part of the document in order to ensure its proper usage. Constraints on accessing the document can be broadly categorized in three classes, namely access control constraints, integrity constraints, and lifecycle constraints.

Access control constraints are the most straightforward. They impose limitations on which data field can be read, modified or written to by each principal. Below, we give examples of access control constraints.

- Some parts of the document cannot be seen by some peers. For instance, the doctor cannot read the insurance number, while the insurance company cannot see medically-sensitive information.
- Some parts of the document cannot be modified by some peers. For instance, while the pharmacist may access the prescription field, he is not allowed to alter it.
- Some actions are not permitted to some peers. For instance, a nurse can never be allowed to write a prescription.
- Other intervening principals can also have limited access to the document. For instance, when testing drugs, researchers could retrieve data on side effects from the file. Also, government officials may access this and other files to report anonymously on the propagation of certain diseases.

Integrity constraints impose restrictions on the values that can be input in each field of the document. Examples of integrity constraints include the requirement that the name of a prescribed drug be selected from a list of government-approved medicines, or the requirement that the total monthly dosage be the sum of every individual daily doses.

Finally, *lifecycle constraints* are restrictions on the ordering in which otherwise valid actions can be performed. If an action occurs "out-of-order", the document will still be readable, but an examination of its history will reveal its inconsistent state. Examples of lifecycle constraints include the requirement that a prescription must be approved by the insurance company before it can

be filled by the pharmacist, or that any test requested by the doctor must be performed before another action is taken on the document.

We suppose that the document can be modified using a small number of atomic actions including `write`, `read` and `update` which can either add a value to the document in a specific field, read a field or update (replace) a value in a given field respectively. The document also supports a number of usage-specific actions, including `perform`, `approve` and `fill`. Action `perform` indicates that a medical test previously requested by the doctor has been carried out, and allows the nurse who has undertaken it to aggregate the information into the test's data structure. The action `approve` is performed by the insurance company employee to indicate that it will reimburse the cost of a prescription. Finally, the `fill` action is performed by the pharmacist upon filling the prescription. In practice, these additional actions are not strictly necessary. Indeed, the same behavior can be stated using the basic actions `write`, `read` and `update`, together with a restriction on which part of the document is being manipulated, however these additional actions will later allow the lifecycle to be stated in a more concise manner.

The lifecycle imposes that no action can be performed on the document before it is initialized, first by filling the personal information section, and then by inputting the insurance number. It is only when these two steps are completed that the document can start to be used to record medical information. We will assume that the document is initially created by the hospital, and initialized for each patient with his personal information and insurance number by the nurse.

After the patient's identifying information has been input, the doctor may take two actions: either prescribe a drug, or request that a medical test be performed. Requesting a test is done using action `test`, which indicates that some value v is written to section *test*. Once this is done, no other action can be taken other than to undertake the requested test, which will be recorded in the document using the specialized action `perform`, indicating that a value containing the results of the last requested test, is written to the `test` data structure. If, on the other hand, the doctor prescribes a drug, it must be first approved by the the insurance company, before it can be filled by the pharmacist. At any step during this process, the patient may report side effects, which are written to the document. If any side effect is reported, no other step may be taken by any principal until a doctor has reviewed the reported side effect using the `read` action.

2.2. Accounting Processes

Another context in which the sequencing of document manipulations is particularly sensitive is banking. In this case, restricting the workflow of document manipulations enforces the proper banking laws and regulations as well as the proper precautions that ensure the prudent management of money. Rao *et al.* [4] recently studied this scenario and proposed a novel formalism for stating the restrictions governing document workflows. Their formalism, the Process Matrix, is strictly more expressive than BPMN as it allows users to place conditional restrictions on the obligation to perform certain steps.

	Activities	Roles			Prede-	Activity
		App	CW	Mgr	cessors	Condition
1	Application	W	R	R		
2	Register Customer Info	W	W	W		
3	Approval 1	W	W	R	$\overset{*}{1,2}$	
4	Approval 2	D	R	W	$\overset{*}{1,2}$	¬Rich
5	Payment	R	W	R	$\overset{* *}{3,4}$	¬Hurry∧ Accept
6	Express Payment	R	W	R	$\overset{* *}{3,4}$	Hurry∧ Accept
7	Rejection	R	W	R	$\overset{* *}{3,4}$	¬Accept
8	Archive	D	W	R	$\overset{* * *}{5,6,7}$	

Figure 1: Process Matrix for a loan application, from [4]

Figure 1 shows the running example they used, which is a loan application process. Each row of the figure represents an activity of the process, listed in the first column. The next three columns indicate the access rights for each of the three roles (applicant, case worker and manager) that a principal can possess in this process. For instance, the applicant can write-out an application, which can then be read by both the case worker and the manager, but only the manager can apply the second approval to a demand for a loan The next column lists the constraints on the sequencing between activities. It distinguishes between regular predecessor, with their usual meaning, and logical predecessor, indicated with an asterisk (*). If activity A is a logical predecessor to activity B, then anytime activity A is re-executed, activity B must also be re-executed. The final column describes optional Boolean activity conditions that may render an activity superfluous. In our example, the second approval can be omitted if the predicate *Rich* holds.

2.3. Data Integrity Policies

The scheme under consideration could also be useful in regards to the enforcement of several classes of Data Integrity policies. All of them can be stated as finite automata [5].

Assured pipelines [6] facilitate the secure transfer of sensitive information over trust boundaries by specifying which data transformation must occur before any other data processing. For instance, assured pipelines can be specified to ensure that confidential data is anonymized before being publicly disseminated, or that user inputs be formatted before being inputted into a system.

A Chinese Wall policy [7] can be set up to prevent conflicts of interest from occurring. For instance, enforcing a Chinese wall policy can prevent a consultant from advising two competing firms, or an investor from suggesting placements in a company in which he holds interest. In this model, a user which accesses a

Figure 2: The Oyster Card used for public transport in London (source: Wikipedia).

data object o is forbidden from accessing other data objects that are in conflict with o.

Sobel *et al.* propose a trace-based enforcement model of the Chinese wall policy, enriched with useful notions of data-relinquishing and time-frames, for which the data management scheme proposed in this paper is suited [8]. In their framework, each object o is associated with a list of action-principal pairs, sequentially listing the actions (either *create* or *read*) each principal performed on the object. On a well-formed object, the list begins with a single *create* event, followed by a series of *reads*. The policy is stated as a set of conflicts of interests $\mathcal{C} \in \mathbb{P}(O)$. Each object O_i is associated with its conflict of interest $\mathcal{C}_i$, that lists the other objects that conflicts with it. The enforcement of the Chinese wall policy is ensured by preventing any user who has accessed a object in set $\mathcal{C}_i$ from accessing object O_i.

Finally, the low-water-mark policy was designed by Biba [9] to capture the constraints that ensure data integrity. In this model, each subject, and each data object, is mapped to a integrity level indicating its *trustworthiness*. A subject can only write to objects that are equal or below its integrity level, and can only read object that are higher or equal of its own integrity level. This prevents subjects and objects from being tainted with unreliable (low-integrity) data.

2.4. Other Examples

The notion of lifecycle policy can be applied to a variety of other domains; we briefly mention some of them in the following.

Smart Cards. Smart cards, such as MIFARE Classic[3], are used to grant access to public transit. They record the number of access tokens their carriers currently hold, as well as a trace of his previous journeys in the system. The card is edited by a *card reader*. Similar cards are used in several contexts including library cards, hotel key cards, membership cards, Social welfare, car rentals and access to amusement parks or museums.

[3] `http://www.mifare.net`

In such a case, the "document" is a physical one, which is carried from one card reader to the next as a passenger travels through the public transit network. However, in this context, the source of mistrust is not the readers, but the carrier of the card. For example, one does not wish the same card to enter twice from the same station, which would likely indicate an attempt at using the same card to get two people in. This is an example of a lifecycle property of the card. The information contained in the MIFARE card could also be used to allow or disallow transfers from one public transit route to another, with any applicable restriction captured in the lifecycle policy.

Sports Data. Johansen *et al.* developed a specific use case of their solution in elite sport teams [10]. Indeed, the impact of sport data analysis on competitiveness is well recognized and implies a growing amount of athlete technical, medical and personal data records. In order to protect these data, *roles* are associated to each category of people wishing to access them and sets of *rules* are defined for each of these roles. These rules can explicit who should have access to what data, but also how these records should be – or *must not* be – manipulated. For example, a coach could be forbidden to access athlete raw medical data but could be allowed to view *smoothed* data over one week. Here, the aim of lifecycle – the set of roles and associated rules – is to ensure athlete data confidentiality and allow them to keep control over their data.

Digital Rights Management. Digital Rights Management (DRM) can be viewed as a special form of user access control that intends to constrain the ways in which a copyrighted document can be used [11]. For example, in the case of a copyrighted picture, one may be interested in limiting the modifications that can be made (cropping, scaling, etc.), and to specify what users are allowed to make these modifications. In the most extreme case, where no modifications to the image are permitted, the only valid lifecycle would be the empty one, meaning that the document should be left unchanged.

3. Enforcing Document Lifecycles

In the Business Process community, constraints on document lifecycles have been studied in the context of "object behaviour models". The most prominent form of such model is artifact-centric business process modelling [12–16]. In this context, various documents (called "artifacts") can be passed from one peer to the next and be manipulated. Rather than (or in addition to) expressing constraints on how each peer can execute, the business process is defined in terms of the lifecycle of the artifacts involved: any sequence of manipulations that complies with the lifecycle of each artifact is a valid execution of the process.

The specification of document lifecycles can be done in various ways. For example, the Business Entity Definition Language (BEDL) [17] allows the specification of lifecycles to business entities as finite-state machines (FSMs). Another possible way of modelling the lifecycle of these artifacts is the Guard-Stage-Milestone (GSM) paradigm [15], as illustrated in Figure 3. This approach

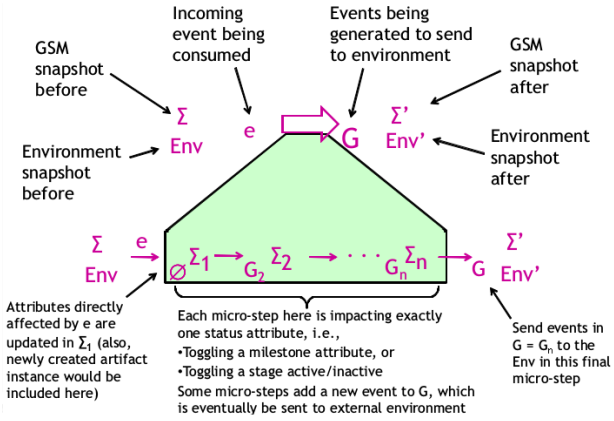


Figure 3: The semantics of the Guard-Stage-Milestone lifecycle paradigm (from [15])

identifies four key elements: an information model for artifacts; milestones which correspond to business-relevant operational objectives; stages, which correspond to clusters of activity intended to achieve milestones; and finally guards, which control when stages are activated. Both milestones and guards are controlled in a declarative manner, based on triggering events and/or conditions. Other approaches include BPMN with data [18] and PHILharmonic flows [19].

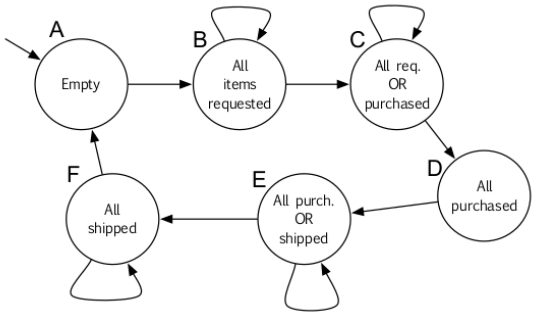
Other specification formalisms borrow heavily from logic and process calculus. For example, Hariri *et al.* study the concept of *dynamic* intra-artifact constraints, and express a finite-state machine-like informal specification into a variant of μ-calculus, as is shown in Figure 4.

In addition, such specification can be compared to expression of *user security and privacy policies*, and more specifically to *sticky policies* [21], which are established by a user or an entity, stick to data and enable the owner to control what operations can be performed on this data when it goes through intermediaries or is shared across multiple service providers. These policies can be expressed with different kinds of policy tags [22, 23] or using XACML [24], a language derived from XML intended to articulate complex privacy policies. All these techniques may be adapted to properly specify artifact lifecycles.

While the specification of artifact lifecycles is relatively well understood, the question of *enforcing* a lifecycle specified in some way has been the subject of many works, which can be categorized as follows.

3.1. Centralized Workflow Approaches

Many works on that topic rely on the fact that the artifacts will be manipulated through a workflow engine. Therefore, the functionalities required to enforce lifecycle constraints can be implemented directly at this central location, since all read/write accesses to the documents must be done through the system. This is the case, for example, of work done by Zhao *et al.* [25]. Similar work



(a) Informal description

$$
\begin{aligned}
\psi_A &= \neg\exists x, y, z.ROItem(x, y, z) \\
\psi_B &= \forall x, y, z.(ROItem(x, y, z) \rightarrow z = requested) \\
&\quad \wedge\ \exists x, y.ROItem(x, y, requested) \\
\psi_C &= \forall x, y, z.(ROItem(x, y, z) \rightarrow (z = requested \vee z = purchased)) \\
&\quad \wedge\ \exists x, y.ROItem(x, y, requested) \wedge \exists x, y.ROItem(x, y, purchased) \\
\psi_D &= \forall x, y, z.(ROItem(x, y, z) \rightarrow z = purchased) \\
&\quad \wedge \exists x, y.ROItem(x, y, purchased) \\
\psi_E &= \forall x, y, z.(ROItem(x, y, z) \rightarrow (z = purchased \vee z = shipped)) \wedge \\
&\quad \exists x, y.ROItem(x, y, purchased) \wedge \exists x, y.ROItem(x, y, shipped) \\
\psi_F &= \forall x, y, z.(ROItem(x, y, z) \rightarrow z = shipped) \wedge \exists x, y.ROItem(x, y, shipped)
\end{aligned}
$$

(b) Formal notation

Figure 4: A dynamic intra-artifact constraint (a) and its formalization into μ-calculus (b); (from [20])

has been done on the database front: Atullah and Tompa propose a technique to convert business policies expressed as finite-state machines into database triggers [26]. Their work is based on a model of a business process where any modification to a business object ultimately amounts to one or many transactions executed on a (central) database; constraints on the lifecycle of these objects can hence be enforced as carefully-written INSERT or UPDATE database triggers.

In contrast, the work we present in this paper does not require any centralized access to the artifacts being manipulated.

3.2. Static Verification

In other cases, knowing the workflow allows to statically analyze it and make sure that all declarative lifecycle constraints are respected at all times [13, 20, 25, 27–29]. For example, Gonzalez *et al.* symbolically represent GSM-based business artifacts, in such a way that model checking can be done on the resulting model [30].

However, verification is in general a much harder problem than preventing invalid behaviours from occurring at runtime; therefore, severe restrictions must imposed on the properties that can be expressed, or the underlying complexity of the execution environment, in order to ensure the problem is tractable (or even decidable). For example, [27] considers an artifact model with arithmetic operations, no database, and runs of bounded length. The approaches in [13, 28] impose that domains of data elements be bounded, or that pre- and post-conditions refer only to the artifacts, and not their variable values [28]. As a matter of fact, just determining when the verification problem is decidable has become a research topic in its own right. For example, Calvanese *et al.* identify sufficient conditions under which a UML-based methodology for modelling artifact-centric business processes can be verified [31].

Furthermore, in a setting where verification is employed, one must *trust* that each peer involved in the process has been statically verified, and also that the running process is indeed the one that was verified in the first place. This hypothesis in itself can prove hard to fulfill in practice, especially in the case of business processes spanning multiple organizations. In contrast, the proposed work eschews any trust assumptions by allowing any peer manipulating an artifact to verify by itself that any lifecycle constraint has indeed been followed by everyone. Moreover, since lifecycle violations are checked at the time of execution (a simpler problem than static verification), our approach can potentially use very rich behaviour specification languages.

3.3. Decentralized Workflow Approaches

The correctness of the sequence of operations can also be checked at runtime, as the operations are being executed; this was attempted by one of the authors in past work [32]. The correctness of the sequence of operations can even be enforced at runtime, still when the operations are being executed; this can be done using techniques and theories (defined for centralized systems) borrowed from runtime enforcement as in e.g., [33, 34] – see [35] for a tutorial. Runtime enforcement has also been suggested, e.g. for the enforcement of lifecycle constraints on RFID tags passing from one reader to the next [36]. Additionally, the LoNet Architecture aims to enforce such lifecycle in the form of privacy policies using meta-code embedded with artifacts [10]. A different approach is to make sure that at runtime, each peer monitors incoming documents or modification requests and check that the constraints are correctly being followed by their respective senders; the constraints are usually expressed using Linear Temporal Logic (LTL).

It is also possible to reuse notions found in decentralized runtime verification and monitoring [37–40]. Runtime monitoring consists in checking whether a run of a given system verifies the formal specification of the system. In this case, the lifecycle is the specification, and the sequence of modifications to the document is the trace to be verified. Decentralized runtime monitoring is designed with the goal to monitor decentralized systems, it is therefore possible to monitor decentralized changes to a document. The approach performs monitoring by progressing LTL —that is, starting with the LTL specification, the monitor rewrites the formula to account for the new modifications. A trace ρ would

comply with a lifecycle expressed as an LTL formula φ if by progressing φ with ρ it results in $\top$. However, at the cost of offering full decentralization, LTL progression could increase the size of the formula significantly as the sequence of actions grows. The growth rate poses a challenge to store the new formula in the document when storage space is small and sequence lengths are large. It is however possible to reduce the overhead significantly by using an automata-based approach [41], at the cost of communicating more between the various components in the decentralized system. This approach in [41] could be suitable for a specific type of lifecycles where interaction is frequent between the various parties.

In a similar way, in *cooperative runtime monitoring* (CRM) [42], a recipient "delegates" its monitoring task to the sender, which is required to provide evidence that the message it sends complies with the contract. In turn, this evidence can be quickly checked by the recipient, which is then guaranteed of the sender's compliance to the contract without doing the monitoring computation by itself. Cooperative runtime monitoring was introduced with the aim of reducing computational load without sacrificing on correctness guarantees. This is achieved by having each peer in a message exchange memorize the last correct state, and use the evidence provided by the other as a quick way of computing the "delta" with respect to the new state —and make sure it is still valid. This differs from the approach presented in this paper in many respects. First, cooperative runtime monitoring expects the properties to be known in advance, and to belong to the NP complexity class; our proposed approach is independent from the lifecycle specification. Second, and most importantly, CRM does not protect the tokens exchanged between a client and a server; a request can be replaced by another, through a man-in-the-middle attack, and be accepted by the server so long as it is a valid continuation of the current message exchange. Finally, the approach is restricted to a single two-point, one-way communication link.

3.4. Cryptographic Approaches

Finally, there are related approaches that are based on security and cryptography. For example, [43] uses Identity-Base Encryption scheme to encrypt cloud-stored artifacts with their own lifecycle as public key. In addition to ensure data confidentiality, it prevents the lifecycle to be tampered as the unchanged private key would not be able to decipher the resulting artifact.

However, in the context of security and cryptography, work on "lifecycle" enforcement has mostly focused on preventing the mediator of the document (for example, the owner of a metro card) from tampering with its contents. Therefore, a common approach is to encrypt the document's content, using an encryption scheme where keys are shared between peers but are unknown to the mediator. This approach works in a context where peers do not trust the mediator, but do trust each other. Therefore, compromising a single peer (for example, by stealing its key) can compromise the whole exchange.

In contrast, our proposed technique provides tighter containment in case one of the peers is compromised. For example, stealing the private key of one of the peers cannot be exploited to force violations of the lifecycle, if the remaining peers

still check for lifecycle violations and deny further processing to a document that contains one. As a matter of fact, we have seen how the peer-action sequence, secured by its digest, can in such a case be used to identify the peer responsible for this deviation of the lifecycle.

The present work can be seen as a generalization of a classical document signature. Indeed, a signature is a special case of this system, where there are only two peers and a single modification operation.

3.5. Blockchain Approach

Bitcoin blockchain is a particular case of centralized workflow approaches and allows peers to transport and store financial *transactions* between them in a distributed way, without any supervisory third-party [44, 45]. The overall goal is to record several cryptographically-protected transactions inside *blocks*, which are then securely chained together using a reference to the previous block of the chain (i.e. a hash of the previous block header). This blockchain can be seen as the open public book of all transactions between peers. Therefore, such system is able to keep track of operations (i.e., adding new transactions) in a specific artifact (i.e., the blockchain), while respecting particular constraints – or a predefined *lifecycle* (i.e., feasibility of transactions and integrity verification of the next block of the chain through the computation of a proof-of-work).

In this context, our proposed approach may be considered as a generalization of blockchains on various aspects. First, the Bitcoin blockchain intends to enforce a predefined lifecycle on transactions and blocks, whereas the present work has the potential to deal with much more general and complex lifecycles. Another point is that blockchain integrity relies essentially on the ability of nodes to compute a proof-of-work in order to add a block to the chain. However, our solution is based on lifecycle enforcement: even if a peer's private key is stolen, it cannot be used to violate the lifecycle as previously stated. Finally, Bitcoin blockchain does not allow a peer to cipher and decipher elements in the name of a group to which he belongs, which is the case in this work.

4. Formalization And Definitions

In this section, we formalize the notions that will be manipulated in Section 5, namely documents, peer-actions, actions, and lifecycles.

In the following, we assume the existence of a hash function $\hbar$. To simplify notation, we assume that the codomain of $\hbar$ is $\mathbb{H}$. We fix P to be the set of *peers*. The set of groups is G and it consists of labels identifying each group. Peers belong to one or more access groups. Groups are akin to the notion of *role* in classical access-control models such as RBAC [46] (see Section 4.3.1): we shall see that belonging to a group gives read/write access to a number of fields of the document under consideration.

4.1. Documents

Let D be a set of documents. A document $d \in D$ is a set consisting of three types of elements:

- Values: a value is a typed data that is referenced by a unique identifier in d. Such identifier will be called a *key*.
- Lists: a list is an array in which each element can refer to another document, such as a value, another list or a map.
- Maps: a map is a set of key-document pairs, where each key refers to another document, such as a value, a list or another map.

Let $\mathcal{V}$ be a set of values, $\mathcal{K}$ a set of keys, $\mathcal{L}$ a set of lists, $\mathcal{M}$ a set of maps. According to the previous definitions, we can represent a list $l \in \mathcal{L}$ as a function $l : \mathbb{N} \to D$ and a map $m \in \mathcal{M}$ as a function $m : \mathcal{K} \to D$. Hence, we define D as a subset of values, lists and maps: $D \subset \mathcal{V} \cup \mathcal{L} \cup \mathcal{M}$. A special document, noted $d_\emptyset$, will be called the *empty document*.

4.1.1. Accessing Elements

The following subsection aims at defining a way to access a specific element – which is itself considered as another document – inside a document d. For this purpose, we characterize a *path* function $\pi : \mathcal{P}^* \times D \to D$ where $\mathcal{P}$ is a set of *path elements*, i.e. integers or keys, $\mathcal{P} \subset \mathbb{N} \cup \mathcal{K}$. In short, π takes a sequence of path elements in a document as input, and returns the corresponding sub-document, provided that the latter exists in the former and that the path is correct.

Formally, we can identify four different expressions for π depending on the nature of the entry inputs. Let $\bar{\sigma} \in \mathcal{P}^*$ be a finite sequence of path elements, we write $\bar{\sigma}' \prec \bar{\sigma}$, to indicate that $\bar{\sigma}'$ is a prefix of $\bar{\sigma}$. Let $\epsilon \in \mathcal{P}$ be the empty path element and $\xi \in D$ the empty document, thus:

$$\pi(\bar{\sigma}, d) \triangleq \begin{cases} \pi(\bar{\sigma}', \pi(n, l)) = \pi(\bar{\sigma}', l(n)) & n \in \mathbb{N}, l \in \mathcal{L} \quad (1) \\ \pi(\bar{\sigma}', \pi(k, m)) = \pi(\bar{\sigma}', m(k)) & k \in \mathcal{K}, m \in \mathcal{M} \quad (2) \\ \pi(\epsilon, d) = d & (3) \\ d_\emptyset & otherwise \quad (4) \end{cases}$$

In the case where d is a list l, given a integer n, then the searched path is in the nth element of l (case (1)). If d is a map, given a key k, the searched path is in the document mapped to k (case (2)). Looping on these two cases, when finally the path becomes empty, meaning that the wanted document was found, π returns the corresponding document (case (3)). However, if at some point the input path is invalid or does not exist in the provided document, e.g., k does not exist in m or n is out of range for l, the function shall return the empty document $d_\emptyset$ (case (4)).

4.1.2. Existing File Formats

Many file formats satisfy our document representation and our path function. Obviously, JSON is one of them since it consists of attribute-value pairs. By using XPath expressions as *path element*, XML is also a good candidate. However, even more readable formats can correspond, such as PDF document. Indeed, using common libraries (such as PDFBox or pdftk), it is possible to assign a unique name to an object when creating a new PDF document. For example, a unique string, used as a *key*, can be mapped to a text field. Doing so allows us to retrieve the value of the field using the specified name, which in our case is considered as our *path* inside the document.

For the sake of clarity, let us consider the following simple JSON document d:

```
{
  "a": {
    "b": [0, 1, 6],
    "c": 9
  },
  "d": 5
}
```

The path function π can be used to retrieve various parts of this document. For example, $\pi(\text{a/b/2}, d) = 6$, since it corresponds to accessing parameter "a" (a map), parameter "b" inside that map (a list), and the third element of that list (assuming list indices start at 0).

4.2. Actions

Let D be a set of *documents* and A be a set of *actions*. Each action $\mathtt{a} \in A$ is associated with a function $f_{\mathtt{a}} : D \to D$ taking a document as an input, and returning another document as its output.

Let AT be a set of possible action types (e.g. `add`, `delete`, `read`). Formally, an action `a` is a 3-tuple $\langle \bar{\sigma}, t, v \rangle \in (\mathcal{P}^* \times AT \times \mathcal{V})$ where $\bar{\sigma}$ is the path leading to the document on which the action is performed, t the type of action performed on the document, and v the new value we want to associate with the targeted document. Note that, for some actions, v can be empty, since some type of actions do not expect any new value. For example, a `delete` action may not need any value because it just erases the current one without replacing it. The same applies for a `read` action that should not modify any value.

We adopt the following notation for representing actions: $\bar{\sigma}(\mathtt{type}, \mathtt{data})$, where $\bar{\sigma}$ is the path of the targeted document, `type` is the action type, and `data` the new data if any. Hence, one could write `patient_number`(`write`, `1A7452N`) to represent the action of assigning the value `1A7452N` to the patient number; here `patient_number` corresponds to the path inside the document leading to the corresponding field. Similarly, writing `side_effects`(`add`, `Nausea`) indicates that the value `Nausea` should be appended to parameter `side_effects`, which is a list. Overwriting the fourth element of that same list by the value `Headache` would be written as `side_effects/3`(`overwrite`, `Headache`).

It should be noted that the scope of this paper is not to define any standard or "good practices": the list of possible types of actions, the possible values for a key and how they should be processed are at the complete discretion of the creator of the document or whoever should be in charge of its management. However, we can distinguish two global kinds of action, described in the next two subsections.

4.2.1. Altering Actions

An altering action (AA) is an action that aims to concretely modify the content and the value of a document. Let $(d, d') \in D^2$, an AA `aa` can be associated to a function $f_{\texttt{aa}}$ such as $f_{\texttt{aa}}(d) = d'$ where $d' \neq d$. This means that the resulting document of an AA is strictly different from the original document.

For example, this kind of action could include `add`, `delete` or `overwrite` actions. An `add` action could be designed to add a new key or value to a document, a `delete` action could erase any value for a document, and an `overwrite` action could modify the content of an existing document. Whereas these actions lead to obvious modifications, there is a possibility that a peer takes an action that should modify a document – an AA – but whose new value is equal to the previous one (e.g., overwrite a document with the exact same current value). This would return a document identical to the input document, and this action could not be considered as an AA anymore. Nevertheless, this kind of action is deemed irrelevant as it does not add any useful information howsoever, we will not consider it for this paper.

4.2.2. Observation Actions

Some actions, however, could provide information without having to alter the body of a document. This kind of actions, that we will call observation actions (OA), only indicates that something has been done on the document. Hence, let $d \in D$, an OA `oa` can be associated to a function $f_{\texttt{oa}}$ such as $f_{\texttt{oa}}(d) = d$.

For instance, any side effects related to a prescribed medication reported by a patient should be approved by his attending physician. In this case, approving a side effect should not modify anything inside the document, but still any peer that needs to check if it has been approved should be able to do it.

A possible way of ensuring this would be to add Boolean fields that could be toggled to *true* when a physician approves a new side effect. While this would work for approval-like actions, it would make other types of read/access actions cumbersome. In the worst case, one might require to keep track of all accesses on all fields of a document, and this could not be easily feasible by simply adding extra fields.

To this end, OAs allow us to process these kinds of actions globally, without having to create additional fields, keys or documents. These accesses are simply tracked within the action history, without any further document modifications. In other words, OAs are just "stamps" added to the peer-action sequence to mark the fact that some peer has seen a particular field of the document in its current state.

4.3. Lifecycles

Before defining what a lifecycle is, we first need to introduce two other elements: peer-actions and peer-action sequences. A peer-action is a 4-tuple $\langle \mathtt{a}, p, g, h \rangle \in (A \times P \times G \times \mathbb{H})$ consisting of an action $\mathtt{a}$, an identifier identifying the peer responsible for this action p and identifying on behalf of which group g was the action taken (the purpose of the hash h will be explained later). We construct a sequence of peer-actions and denote it by $\overline{s}$. The set S contains all possible peer-action sequences.

A document *lifecycle* specifies what actions peers are allowed to make on a document and in which order. It is represented by a function $\delta : S \to \{\top, \bot\}$. Intuitively, the function δ takes as input a peer-action sequence, and decides whether this sequence is valid ($\top$) or not ($\bot$).

A *lifecycle* can be seen as sets of constraints that a peer-action sequence must respect. We classify these constraints into three categories: access, integrity and order constraints.

4.3.1. Access Constraints

Since fields of competences and responsibilities can be many and varied inside a company or an institution, it seems logical that there might be as many and varied authorization levels and access control permissions for a document. For example, an engineer might be authorized to fill the technical details of a project, but only the project manager should be able to sign the document. Also, the engineer might not be allowed to read some financial details he does not need to know, whereas the project manager should have access to the whole document. These constitute the access constraints.

As stated in the introduction of Section 4.3, a peer always takes an action on behalf of a group, which obviously requires that the peer in question must be part of this specific group. This means that peers have different access rights depending on which group they take the action for. Thus, in order to determine if a peer $p \in P$ is authorized to perform an action $\mathtt{a} \in A$ on behalf of a group $g \in G$, we must assess if p really belongs to g and if members of g have the rights to perform $\mathtt{a}$. Membership can be evaluated using the predicate $M : P \times G \to \{\top, \bot\}$, $M(p, g) = \top$ meaning that the peer belongs to the group, and $\bot$ meaning he does not. Group's permission is assessed using the function $access : G \times A \to \{\top, \bot\}$, $access(g, \mathtt{a}) = \top$ meaning that members of g are allowed to perform $\mathtt{a}$, and $\bot$ meaning they are not.

For each group $g \in G$ we associate the *group lifecycle* function δ_g and we assign peers to groups using the predicate $M(p, g)$. Function δ_g specifies the actions allowed for a member of the group to make on the document. A peer $p \in P$ belongs to the set of groups $G_p = \{g \mid M(p, g) = \top\}$. The lifecycle that p will verify is $\delta_p(s) : S \to \{\top, \bot\}$, with $\delta_p(s) = \bigwedge_{g \in G_p}(\delta_g(s))$, where $\top$ and $\bot$ are interpreted as Boolean `true` and `false` respectively. The lifecycle δ_p ensures that p can only verify the lifecycles of groups they belong to. We add the restriction that when a peer executes an action on a document, they execute it on behalf of one group only. In this case, note that a group lifecycle δ_g acts

on the entire sequence. Therefore, the specification must be written in a way that δ_g is only concerned with the actions relevant to the group, ignoring the rest of the sequence and handling synchronization.

RBAC. Obviously, peers are part of different and potentially multiple groups. Groups and accesses could be managed using mechanisms such as Role-Based Access Control (RBAC) [46]. Peers would be assigned a *role* and this one could be mapped to several groups. For example, the role *engineer* would be mapped to the groups *Engineers* and *Employees*, the role *accountant* to the groups *Accountants* and *Employees*, and the role *project manager* to the groups *Project Managers*, *Engineers*, *Accountants* and *Employees*. This means that a project manager would have the combined access rights of accountants and engineers on a document, plus those of his own group.

ABAC. Other mechanisms could be involved such as Attribute-Based Access Control (ABAC) [47]. Instead of mapping existing *roles* to groups, a particular group membership could be determined through attributes like the peer's position, department or current projects.

4.3.2. Integrity Constraints

Documents may have internal constraints on their content that could be based on their *format*. For example, a field in a PDF document might be expected to contain only text and its size to be bounded, whereas an attribute in a JSON object could be intended to contain an array of integers exclusively. Moreover, document values might be related to one another under some conditions. For instance, a document may include a key *drug price* holding the price of a drug bought by a patient, and a key *reimbursement amount* holding the amount that an insurance is willing to refund for that specific drug. In this case, the *reimbursement amount* could not be higher than the *drug price* or a certain percentage of its value.

On a more general note, integrity constraints are restrictions on the fields of a document, they define out of all possible documents, a valid subset $D_{\text{valid}} \subseteq D$. Given a document $d \in D_{\text{valid}}$ and an action $\mathtt{a}$, checking the integrity of the document after applying $\mathtt{a}$ amounts to computing $d' = f_{\mathtt{a}}(d)$ and checking $d' \in D_{\text{valid}}$. If $d' \notin D_{\text{valid}}$, then the action violated the integrity constraints. Checking an entire sequence is done by successively applying the check after each action ensuring the resulting document remains valid.

In order to express and enforce these constraints, we can draw on existing solutions like XACML [24] or WSDL [48]. These two XML-based languages allow to describe *policies*, i.e., request-response management for XACML and web service definition for WSDL, and could be adapted to match our case. However, two languages, also based on XML, fit perfectly to our situation: XML DTD and XML Schema. JSON Schema is a good candidate as well since it is roughly an adaptation of XML Schema, the main difference being that the latter is designed for XML documents and the former for JSON documents, which is why we will not elaborate on it.

XML Document Type Definitions. A Document Type Definition (DTD) file allows to describe an expected structure for a XML document. DTD offers a whole range of possibility to express format constraint. For example, we create a `drug_prescription.dtd` file containing a very simple but possible structure for a drug prescription:

```
<!ELEMENT drug_prescription (date,doctor_name,patient_nb,side_effects)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT doctor_name (#PCDATA)>
<!ELEMENT patient_nb (#PCDATA)>
<!ELEMENT side_effects (side_effect+)>
<!ELEMENT side_effect (type,hours_since_prescription)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT hours_since_prescription (#PCDATA)>
```

The first line of this listing stipulates that an element called `drug_description` should be composed of elements named `date`, `doctor_name`, `patient_nb`, and `side_effects`. In turn, the second line describes that `date` is composed of a character string (`#PCDATA`), and so on for the remaining lines. The combination of all these declarations precisely describes the possible structure of the document. Any XML document respecting this format is a correct *drug prescription* as defined in the DTD file. For example, the following file is correct:

```
<?xml version="1.0"?>
<!DOCTYPE drug_prescription SYSTEM "drug_prescription.dtd">
<drug_prescription>
  <date>2017-05-02</date>
  <doctor_name>Dr. Meredith Grey</doctor_name>
  <patient_nb>1A7452N</patient_nb>
  <side_effects>
    <side_effect>
      <type>Headache</type>
      <hours_since_prescription>2</hours_since_prescription>
    </side_effect>
    <side_effect>
      <type>Nausea</type>
      <hours_since_prescription>3</hours_since_prescription>
    </side_effect>
  </side_effects>
</drug_prescription>
```

XML Schema. XML Schema, also referred as XML Schema Definition (XSD), is really similar to DTD. Besides, it is even possible to cross XSD and DTD files. However, with XSD it is possible to specify types to elements (e.g. Boolean, date, integer) and restrictions such as enumeration (i.e. the value has to be among a list of possible value), pattern (i.e., the value must respect a regular expression) and many others. The same drug prescription structure previously used can be written with XSD:

```
<xs:element name="side_effect">
```

```
  <xs:complexType>
    <xs:element name="type" type="xs:string"/>
    <xs:element name="hours_since_prescription" type="xs:integer"/>
  </xs:complexType>
</xs:element>

<xs:element name="side_effects">
  <xs:complexType>
    <xs:element ref="side_effect" maxOccurs="unbounded"/>
  </xs:complexType>
</xs:element>

<xs:element name="drug_prescription">
  <xs:complexType>
    <xs:element name="date" type="xs:date"/>
    <xs:element name="doctor_name" type="xs:string"/>
    <xs:element name="patient_nb" type="xs:string"/>
    <xs:element ref="side_effects"/>
  </xs:complexType>
</xs:element>
```

The sample XML document shown earlier is also valid according to this schema.

4.3.3. Order Constraints

Actions performed on a document might be expected to follow a chronological order. For example, if a document has to be signed, it should be only provided that all of its sections have been filled beforehand. Moreover, the processing of its different sections might also have to match a specific sequence since some of them could refer to previous ones. Whatever the reason, order constraints are widely used, especially in business process management, and form the basis of actual document lifecycles.

Such constraints can be specified in various ways. Section 3 depicts existing methods such as FSMs, GSM paradigm or BPMN. However, Petri nets, Linear Temporal Logic (LTL), and any other model or language describing a predetermined sequence of actions could be used.

To adapt order constraints to our formalization, we note that assessing the compliance of a peer-action with order constraints relies only on the action performed, not on the peer or the group (see Section 4.3.1 for these matters). Even more precisely, only the type and the targeted document of the actions involved are decisive (see Section 4.3.2 for data integrity). Let, for some natural number n, $\overline{\mathtt{a}} = (\mathtt{a}_0, \mathtt{a}_1, \ldots, \mathtt{a}_n)$ be a finite sequence of actions, and $\mathtt{a}_{n+1}$ a new action, the function $order : A^* \times A \to \{\top, \bot\}$ evaluates if a new action respects the order constraints in regards to previous actions, i.e., $order(\overline{\mathtt{a}}, \mathtt{a}_{n+1}) = \top$, or not, i.e. $order(\overline{\mathtt{a}}, \mathtt{a}_{n+1}) = \bot$.

In the next paragraphs we will show how we can adapt some of the previously-mentioned methods to express order constraints.

Finite-State Machines. A finite-state machine (FSM) contains a set of states in which events *trigger* the transition from a state to the next one. The set of possible events is called an *alphabet*. In our case, the *alphabet* may only consist of strings, each representing a tuple $\langle \bar{\sigma}, type \rangle \in (\mathcal{P}^* \times AT)$ involving the targeted document and the action type. Let F be a FSM, Σ its alphabet, Q its set of states, $q_0 \in Q$ its initial state and $q_{invalid} \in Q$ a final state. We note $\varphi : A \to \Sigma$ the function that takes an action as input and transforms it into a valid element of the alphabet based on the action key and its type. Starting from q_0, we process the very first action $\mathtt{a_0}$: if $\varphi(\mathtt{a_0})$ corresponds to a possible transition from q_0, then the action is valid and we move to the next state ; if not, we move to $q_{invalid}$. Thus, as actions are processed successively and we are moving through the valid or invalid states, we can determine if these actions are valid or not. This formalization allows us to specify a sequential order based on the type of action performed and the document in question, which is exactly what we expect from order constraints as we have defined them in this section. We note that the same method can be used to adapt Petri nets so that they can process actions, the difference being that the function φ should be $\varphi : A \to T$ where T is the set of transitions in the Petri net.

LTL. Linear temporal logic (LTL) is a modal temporal logic that can be used to evaluate if a specific trace respects some conditions in terms of sequencing. These conditions are combined with classical logical operators and temporal modal operators to form *formulæ*. Let $\mathtt{a^i} = (\mathtt{a_i}, \mathtt{a_{i+1}}, \ldots, \mathtt{a_n})$ be a suffix of $\bar{\mathtt{a}}$. The fact that $\bar{\mathtt{a}}$ satisfies a given formula ψ is noted $\bar{\mathtt{a}} \models \psi$. In the present context, the ground terms of an LTL formula are tuples of the form $\langle \bar{\sigma}, type \rangle \in (\mathcal{P}^* \times AT)$. An element e of a peer-action sequence satisfies the ground term $\langle \bar{\sigma}, type \rangle$ if its action targets $\bar{\sigma}$ and is of type *type*.

Ground terms can be combined with the classical Boolean connectives $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not") and $\to$ ("implies"), following their classical meaning. In addition, LTL *temporal operators* can be used. The temporal operator $\mathbf{G}$ means "globally". For example, the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every event of the trace, starting from the current event. The operator $\mathbf{F}$ means "eventually"; the formula $\mathbf{F}\,\varphi$ is true if φ holds for some future event of the trace. The operator $\mathbf{X}$ means "next"; it is true whenever φ holds in the next event of the trace. Finally, the $\mathbf{U}$ operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if φ holds for all events until some event satisfies ψ. The formal semantics of LTL is summarized in Table 1.

For example, to express the fact that an action $\langle \bar{\sigma}, t \rangle$ cannot occur until $\langle \bar{\sigma}', t' \rangle$, one can write $\mathbf{G}(\neg \langle \bar{\sigma}, t \rangle\,\mathbf{U}\,\langle \bar{\sigma}', t' \rangle)$.

5. Lifecycle Enforcement With Peer-Action Sequences

To alleviate the issues mentioned in Section 3, we describe in this section an original technique for storing a history of modifications directly into a document. Given guarantees on the authenticity of this history (which will be provided through the use of hashing and encryption), this technique allows any peer

$$
\begin{array}{rcl}
\bar{\mathtt{a}} \models \sigma & \equiv & \bar{\mathtt{a}}[0] = \sigma \\
\bar{\mathtt{a}} \models \neg\varphi & \equiv & \bar{\mathtt{a}} \not\models \varphi \\
\bar{\mathtt{a}} \models \varphi \wedge \psi & \equiv & \bar{\mathtt{a}} \models \varphi \text{ and } \bar{\mathtt{a}} \models \psi \\
\bar{\mathtt{a}} \models \mathbf{G}\,\varphi & \equiv & \bar{\mathtt{a}}[i] \models \varphi \text{ for all } i \\
\bar{\mathtt{a}} \models \mathbf{X}\,\varphi & \equiv & \bar{\mathtt{a}}_1 \models \varphi \\
\bar{\mathtt{a}} \models \varphi\,\mathbf{U}\,\psi & \equiv & \bar{\mathtt{a}} \models \psi, \text{ or both } \bar{\mathtt{a}} \models \varphi \text{ and } \bar{\mathtt{a}}[1..] \models \varphi\,\mathbf{U}\,\psi
\end{array}
$$

Table 1: Semantics of LTL

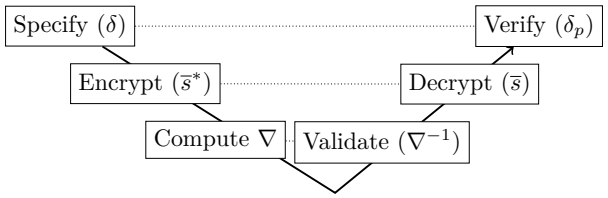


Figure 5: Lifecycle Enforcement

to retrieve a document, check its history and verify that it follows a lifecycle specification at any time.

In the following, we assume the existence of public key encryption/decryption functions; the notation $E[M, K]$ designates the result of encrypting message M with key K, while $D[M, K]$ corresponds to decryption. Each peer $p \in P$ possesses a pair of public/private encryption keys noted $K_{p,u}$, $K_{p,v}$, respectively. We decentralize the specification by incorporating different groups and for each $g \in G$ we consider a symmetric key S_g.

Figure 5 illustrates our general approach to enforcing lifecycles. First we begin by defining the lifecycle δ, then show how a sequence can be encrypted to hide information from various groups, and how its digest is computed to ensure its integrity. Moreover, we explain how the sequence can be verified given its digest, then decrypted and verified by every peer p based on their permissions (δ_p).

5.1. Encrypting a Sequence

Before storing the peer-action sequence in the document, we ensure confidentiality for group actions. For the scope of this paper, we seek to disallow non-group members to see which exact action has been taken, but not the fact that an action has been taken. A peer action $\langle \mathtt{a}, p, g, h \rangle$ where peer p has taken an action $\mathtt{a}$ on behalf of g is encrypted as $\langle E[\mathtt{a}, S_g], p, g, h \rangle$. The actual peer-action sequence stored in the document is $\bar{s}^* : (P \times \mathbb{H} \times G \times \mathbb{H})^*$. This ensures that members outside the group can see that the peer p has taken an action on behalf of the group g (thus are able to check $M(p, g)$), but cannot see which action ($\mathtt{a}$)

has been taken. Therefore they cannot know which $f_{\mathtt{a}}$ has been applied to the document.

5.2. Computing a Digest

The enforcement of a lifecycle is done by calculating and manipulating an history *digest*.

Definition 1 (Digest). *Let $\overline{s}^* = (pa_1^*, \ldots pa_n^*)$ be an encrypted peer-action sequence of length n, and let $\overline{s'}^* = (pa_1^*, \ldots, pa_{n-1}^*)$ be the same sequence, trimmed of its last peer-action pair, where $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ for $i \in [0, n]$. The digest of $\overline{s}^*$, noted $\nabla(\overline{s}^*)$, is defined as follows:*

$$\nabla(\overline{s}^*) \triangleq \begin{cases} 0 & \text{if } n = 0 \\ E[\hbar(\nabla(\overline{s'}^*) \cdot a_n^* \cdot g_n), K_{v,p_n}] & \text{otherwise} \end{cases}$$

In other words, to compute the n-th digest of a given encrypted sequence $\overline{s}$, the peer p_n responsible for the last action $\mathtt{a_n}$ on behalf of the group g takes the last computed digest, encrypts $\mathtt{a_n}$ with the group key S_g, concatenates $E[a_n, S_g] \cdot g$, computes its hash, and encrypts the resulting string using its private key K_{v,p_n}. A side effect of signing is that it ensures that the content to be encrypted is of constant length, and the signature does not expand as new actions are appended to the document's history. Signing with the group id appended to the action is used to ensure the integrity of the group advertised.

The digest depends on the complete history of the document from its initial state. Moreover, each step of this history is encrypted with the private key of the peer having done the last action. Note that encrypting each tuple of the history separately would not be sufficient. Any peer could easily delete any peer-action pair from the history, and pretend some action did not exist. In the same way, a peer could substitute any element of the sequence by any other picked from the same sequence, in a special form of "replay" attack. Adding the action's position number in the digest would not help either, as any suffix of the sequence could still be deleted by anyone. Moreover, in this scheme, forging a new digest requires knowledge of other peers' private keys.

5.3. Checking a Digest

In addition to its data, a document should also carry the encrypted peer-action sequence and a corresponding digest. Checking that the sequence corresponds to the digest is done by verifying group membership and the digests over the entire sequence.

Definition 2 (Verify Digest). *Given an encrypted peer-action sequence $\overline{s}^* = (pa_1^*, \ldots, pa_n^*)$ of length n, and a digest d. Let $\overline{s'}^* = (pa_1^*, \ldots, pa_{n-1}^*)$ be the same sequence, trimmed of its last peer-action pair, and $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ for $i \in [0, n]$. The sequence $\overline{s}^*$ verifies d if and only if $\nabla^{-1}(\overline{s}^*, d) = \top$, where:*

$$\nabla^{-1}(\overline{s}^*, d) \triangleq \begin{cases} \nabla^{-1}(\overline{s'}^*, h_{n-1}) \wedge M(p_n, g_n) \wedge & \textit{if } n > 0 \\ \quad D[h_n, K_{p_n,u}] = \hbar(h_{n-1} \cdot a_n^* \cdot g_n) & \\ \top & \textit{otherwise} \end{cases}$$

To check the digest, the digest is decrypted with the public key of p_n. This results in the computed hash of the sequence. We verify afterwords that the action a_n^* and group g_n are authentic by recomputing the hash and checking it against the signed hash. Detecting a fraudulent manipulation of the digest or the peer-action sequence can be done in the following ways:

1. computing $D[h_n, K_{p_n,u}]$ produces a nonsensical result, indicating that the private key used to compute that partial digest is different from the one advertised, therefore the peer authenticity is compromised;

2. computing $D[h_n, K_{p_n,u}]$ produces a different outcome than would $\hbar(h_{n-1} \cdot a_n^* \cdot g_n)$, invalidating the authenticity of the advertised action and group;

3. observing that p_n is not in g_n.

We note that even if the action is hidden, it is still possible for any peer to verify that, at the very least, peer p_n belongs to g_n and knows that p_n has taken an action.

5.4. Decrypting a Sequence

Once a peer validates the authenticity of a sequence, the peer will then have to decrypt the sequence to process the actions. The decryption of an encrypted peer-action sequence depends on what the peer can see. The new sequence will depend on the groups the peer belong to.

Definition 3 (Decrypting a Sequence). *Given an encrypted peer-action sequence $\overline{s}^* = (pa_1^*, \ldots, pa_n^*)$ of length n. Let $\overline{s'}^* = (pa_1^*, \ldots, pa_{n-1}^*)$ be the same sequence, trimmed of its last peer-action pair, and $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ for $i \in [0, n]$.*

$$\mathrm{SD}(\overline{s}^*, p) \triangleq \begin{cases} \mathrm{SD}(\overline{s'}^*, p) \cdot pa_n & \textit{if } M(p, g_n) \wedge n > 0 \\ \mathrm{SD}(\overline{s'}^*, p) & \textit{if } \neg M(p, g_n) \wedge n > 0 \\ \epsilon & \textit{otherwise} \end{cases}$$

where $pa_n = \langle D[a_n^*, K_{g_n}], p_n, g_n, h_n \rangle$ and ϵ is the empty sequence.

In the case where the peer belongs to the group advertised ($M(p, g_n)$), the last action is decrypted using the group key ($D[a_n^*, K_{g_n}]$) and the resulting tuple is included in the decrypted sequence. Otherwise, when the peer does not belong to the group ($\neg M(p, g_n)$), the entire tuple is discarded from the sequence.

5.5. Checking the Lifecycle

A peer p verifies the lifecycle of the document based on his groups. To do so, the peer first computes $\overline{s}_p = \mathrm{SD}(\overline{s}^*, p)$, then ensures that $\delta_p(\overline{s}_p) = \top$. $\overline{s}_p$ is the sequence that p can decrypt based on his groups, while δ_p is the lifecycle p can verify[4].

5.6. Checking the Document

The purpose of the digest is to provide the receiver of a document a guarantee about the authenticity of the peer-action sequence that it contains. This sequence, in turn, can be used to check that the document being passed is genuine and has not been manipulated.

Given the decrypted peer-action section for p denoted by $\overline{s}_p = \mathrm{SD}(\overline{s}^*, p) = (\langle \mathtt{a}_0, p_0, g_0, h_0 \rangle, \ldots, \langle \mathtt{a}_k, p_k, g_k, h_k \rangle)$. Since the peer-action sequence can omit some encrypted parts, we have $|\overline{s}_p| \leq |\overline{s}^*|$. Starting from the base document $d_\emptyset$ it is possible to compute the new document $d = f_{\mathtt{a}_k}(f_{\mathtt{a}_{k-1}} \cdots (f_{\mathtt{a}_1}(d_\emptyset)) \cdots)$, and compare it with the document being passed. In other words, it is possible for a peer to "replay" the complete sequence of actions, starting from the empty document, and to compare the result of this sequence to the actual document. Since some actions are hidden from the peer, it is not possible to reconstruct the entire document unless p is in all groups (in which case $\overline{s}_p = \overline{s}$). However, it is possible to verify a part of the document if we consider the following assumptions on the specification:

1. The data in the document is partitioned into pairwise disjoint sets $D = \bigcup_{g \in G}(D_g)$.
2. For each action a appearing in a lifecycle δ_g, f_a either modifies data in one set of data or no data at all (i.e., it does not modify the document).

With these assumptions, people in the group can "replay" only the data relevant to the group, since group actions do not interfere with it. Actions associated with functions that do not modify the document could be used to synchronize the various groups. One could consider that each set of fields is encrypted with the group key, so as to not be visible to other groups.

Note that in some cases, knowledge of the peer-action sequence and of the empty document is sufficient to reconstruct the complete document without the need to pass it along. In such cases, only exchanging the sequence and the digest is necessary. However, there exist situations where this does not apply —for example, when the document is a physical object that has to be passed from one peer to the next (as in the case of a metro card), or when the data subject to modification is a subset of all data carried in the document.

[4]This checking is possible for any implementation of δ, we provide an automata based formalization for δ in Section 5.8.

5.7. Multigroup Actions

Since peers can belong to multiple groups, actions also can be of interest to several groups. Thus, they should be visible to their groups of interest. Peers in one group but not another must be able to see it. Therefore, in our current approach, to share an action $\mathsf{a}_{\text{shared}}$ with n groups, it is necessary to append the action to the peer-action sequence n times. Each instance is encrypted with the symmetric key of one group. On the one hand, this could lead to inefficiency as the action is duplicated several times. On the other hand, it forces restrictions on the lifecycle checking function δ. If a peer p belongs to two or more groups that share $\mathsf{a}_{\text{shared}}$, then they will respectively have two or more repetitions of $\mathsf{a}_{\text{shared}}$ in the decrypted sequence. Thus, to be able to correctly check it, δ must account for the repetitions in that its output should be insensitive to repetitions, and thus must be stutter invariant [49].

To avoid these issues, we recommend using new groups for shared actions between groups. We call these groups supergroups. We create a new group g_{super}, with all the peers involved, and sign a_{shared} with $K_{g_{\text{super}}}$. Furthermore, this allows us to specify the behavior of the shared actions using $\delta_{g_{\text{super}}}$. We note that the shared group includes all members of the other groups. Therefore, it suffices to encrypt the action with $K_{g_{\text{super}}}$, and it will appear in the trace of all involved peers.

5.8. Implementing δ as Extended Automata

To assemble the specification given all constraints expressed in Section 4.3, we express the lifecycle using a set of automata (one for each group). The alphabet of the automaton of a group g is a subset of the actions: $\Sigma_g \subseteq A$. Σ_g is partitioned into two sets: $\Sigma_g = L_g \cup B_g$. L_g contains *local* actions, that is, actions undertaken by the group g. B_g contains *border* actions, that is, actions undertaken by other groups but shared with g (they are multigroup actions, see Section 5.7). Border actions serve as synchronization actions between groups. A function $\mathcal{S}_g : B_g \to 2^G$ assigns the border actions to a set of groups that could emit them, for verification purposes. Each group has an automaton $\mathcal{A}_g = \langle Q_g, \Sigma_g, \Delta_g, q_g^0, F_g, \mathcal{S}_g \rangle$ where: Q_g is a set of states, $\Delta_g : Q_g \times \Sigma_g \to Q_g$ is the automaton transition function, q_g^0 is the initial state, F_g is a set of accepting states. Additionally, we add a sink state $q^{\text{fail}} \notin F$ such that $\forall s \in \Sigma_g : \Delta(q^{\text{fail}}, s) = q^{\text{fail}}$. We extend the transition function Δ_g to Δ'_g to account for verifying peer action sequences:

$$\Delta'_g(q, \langle a', p', g', h' \rangle) \triangleq \begin{cases} q' & \text{if } \Delta_g(q, a') = q' \\ & \quad \wedge((a' \in L_g) \vee (a' \in B_g \wedge \exists g'' \in \mathcal{S}_g(a') : M(p, g''))) \\ q & \text{if } a' \notin \Sigma_g \\ q^{\text{fail}} & \text{otherwise} \end{cases}$$

Starting from a state q and given an authenticated and decrypted peer action $\langle a', p', g', h' \rangle$, we first verify the integrity constraints on a' then check the next state. If the action is a border action, we make sure that the peer performing the

action is doing it belongs to a group allowed to emit it by $\mathcal{S}_g$. In the case where the action is not related to the group specification $\mathcal{A}_g$, that is, it is not in the alphabet Σ_g, we simply ignore it. Running a peer action sequence $(pa_0, \ldots, pa_n)$ can then be done as follows:

$$\Delta_g^+(q, (pa_0, \ldots, pa_n)) \triangleq \begin{cases} \Delta_g^+(\Delta_g'(q, pa_0), (pa_1, \ldots, pa_n)) & \text{if } n > 0, \\ \Delta_g'(q, pa_0) & \text{otherwise.} \end{cases}$$

Checking a peer action sequence for a group can be done as follows:

$$\delta_g(s) \triangleq \begin{cases} \bot & \text{if } |s| > 0 \wedge \Delta_g^+(q_g^0, s) \notin F_g \\ \top & \text{otherwise} \end{cases}$$

By running the peer-action sequence using Δ_g^+ starting from the initial state (q_g^0) and reaching a non-accepting state, we conclude that the lifecycle has been violated $(\delta_g(s) = \bot)$.

5.9. Handling Dynamic Groups

A document lifecycle is not always small, one issue that arises when handling long lifecycles is that groups may be modified while the document is still circulating. Therefore, we discuss an approach to account for the change in groups. A direct approach involves creating a new group whenever membership is modified, that is, whenever a user leaves or joins a group. In this approach, the new group inherits the sub-specification of the old group, and the membership predicate $\mathrm{M}(p, g)$ is updated accordingly to account for the new group.The new group will be assigned a new symmetric key[5]. A new key ensures that the user joining the group cannot see the history of actions taken by the group prior to the join, and will no longer see any actions made by the group after leaving. Therefore, this ensures that all actions in the lifecycle remain valid after membership changes.

However, since all possible groups cannot be accounted for a priori, it is not possible to account for changes before they happen in the membership predicate. One solution is to centralize the membership information, making it available for all parties in a centralized manner. Thus, groups and the member assignments are modified in one location, and peers simply query the centralized point to check.

Another more flexible solution is to start with initial information of members and groups, and update it using specific actions. We propose two actions: `join` and `leave`. Since group membership is public, we require that a public group $g_{\text{pub}} \in G$ exists.

While we discuss group membership as public in this paper, it is also possible to create secret groups. In this scenario, memberships changes are not broadcast to a public group, but to any group that is interested in knowing about the

[5]The new key can be negotiated using existing key-exchange protocols [50].

changes. However, this limits the possibility to validate the lifecycle for group memberships to only groups that the peer is in.

Modifying the membership of p in a group g requires a new action to be placed on the lifecycle. We define two actions: $\langle \texttt{join}, p, g, g' \rangle$ and $\langle \texttt{leave}, p, g, g' \rangle$ to indicate a join and leave, respectively. In the case of a join, since the user is initially not in the joined group, we require that another peer p' from g (i.e., a peer p', such that $\mathrm{M}(p', g) \wedge p \neq p'$) submits the action. When appended to the lifecycle, the action is signed with the public group key $K_{g_{\mathrm{pub}}}$, so as to advertise it to everyone. We note that, since the membership modification is presented as an action for a person in a group, the behavior can be checked for compliance with the public group specification (δ_{pub}). Since all peers belong to g_{pub}, the action will be always present in the decrypted sequence. Thus, it is also possible to define per-group policies as to who can allow people to join or leave a certain group. In the case of a join, since the action is submitted on behalf of p by another peer p', it is also possible to require that the action be also signed by p as a confirmation.

Upon modifying the group in any of the two solutions presented above, a new group g' is created with $\delta_{g'} = \delta_g$, consisting of all members of g with the addition or removal of p.

5.10. Fast-Forwarding and Deleting Prefixes

As such, the peers in the exchange can be completely *stateless*: they are not required to persist any information between accesses to a document, apart from their public/private key pair.[6] All the history and the verification of the lifecycle can be reconstructed from the empty document at any time.

However, since the history lengthens over time, the total processing over the lifetime of the document will be quadratic in the length of its history. On the other hand, a stateful peer can save on processing time: for each document, such a peer can save the digest and state of the document each time it receives it. Upon receiving it another time, it only requires to invert the digest and check the document's contents up to its last locally-saved state. (This is possible, since the probability for a tampered document to yield the same n-th digest as the original is very small.) This way, each element of the peer-action sequence requires processing only once.

We now present a mechanism allowing a peer p to "freeze" a part of the peer-action sequence, in such a way that it will not need to be rebuilt or re-verified. To this end, we introduce a special idempotent action, (ι_p, k), where ι is a dummy name and k is an arbitrary data element. The purpose of action ι is to allow a peer performing this action to "save" into the document a snapshot of its current contents, as well as any piece of the peer's internal state that needs to be

[6] They must also remember the lifecycle function being enforced; however this could even be saved within the document and encrypted with their private key. In any case the function is likely to be the same for all documents, and could in some cases be hard-coded into the peers' read-only memory.

retrieved in order to resume the verification of the lifecycle from the appropriate point. For the purpose of lifecycle constraints, ι actions are simply ignored, as they do not represent actual manipulations to the document.

Technically, a peer p that wishes to save such a snapshot into the peer-action sequence simply replaces a regular action a by a pair (ι_p, k), and does not encrypt this pair using a group key. The computation of the digest (hashing and encryption with the peer's private key) is then done as usual. Malicious tampering with the contents of k can be detected by checking the digest, as for any other action.

When p receives a sequence, it can read it backwards and check it as usual; however, this validation can be stopped as soon as it encounters an action ι_p. If decrypting the associated value k returns a string that is deemed valid, p can safely assume that the validation of the prefix of the trace up to that point has already been done, and retrieve from k the contents of the associated document. Moreover, if k contains information about p's internal state, it can be used to restore p to the state it was in at that point in the peer-action sequence. The evaluation of the lifecycle policy can then be resumed from that point up to the end of the sequence.

In this way, ι actions allow a peer to "fast-forward" the evaluation of the trace to the latest checkpoint saved into the sequence. Thus, even if a peer does not persist the state of the document and its associated lifecycle policy between accesses, the evaluation does not need to be done from the start every time.

This mechanism can be made more powerful if peers are allowed to keep a persistent memory of the document's content. Instead of saving the contents of the document into a pair (ι_p, k), a peer can instead write an action (ι'_p, k'), where k' is a sequential number encrypted with p's private key. The purpose of such an action is to indicate that p has kept a *local* snapshot of the document at this point, and that it was deemed valid up to that point according to its lifecycle policy.

When a peer receives a sequence, it can look for the largest position i such that for every peer p, there exists an action (ι'_p, k') at position j for some $j \geq i$. This represents the latest point in the sequence that has been kept in local memory by all peers. When re-transmitting the document to other peers, the prefix of the sequence up to position i can be deleted. Attempting to delete more than that prefix can be detected by at least one peer, as each keeps in memory the last action (ι'_p, k') they added to the trace. Receiving a sequence that does not contain it indicates that its validity can no longer be assessed by that peer.

6. Use Case Revisited

We shall now revisit the use case of Section 2.1, and illustrate how the informal constraints exposed there can be expressed formally in terms of the concepts introduced in this paper. We focus on the earlier part of the use-case which consists in the filling of the patient profile by the nurse, and approval from the insurance company. We will use the following peers: a doctor (d), a

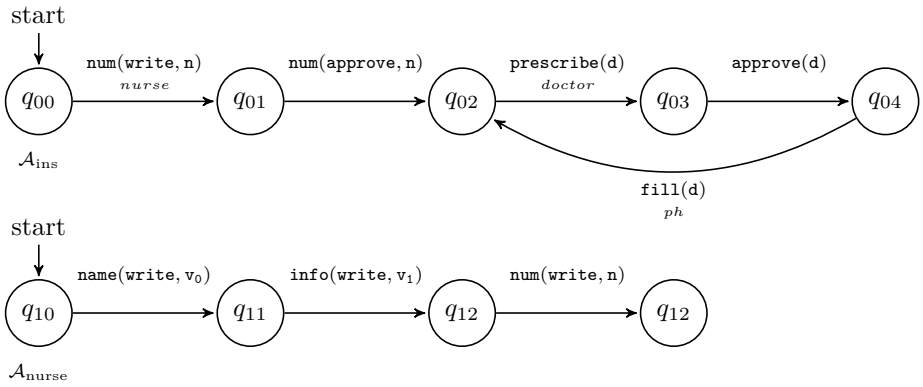


Figure 6: Partial Lifecycle example

nurse (n), the insurance company (i) and the pharmacist (r). The pharmacist is used to illustrate privacy in this scenario. For that we consider the following groups: $G = \{\texttt{nurse}, \texttt{ins}, \texttt{ph}, \texttt{doctor}, \texttt{nurse/ins}, \texttt{hosp}, \texttt{pub}\}$. The first four groups represent the groups for nurses, insurance, pharmacists, and doctors. For this example, each of these groups contains only one relevant peer. The first four groups consist respectively of n, i, r, and d. The group `nurse/ins` is a supergroup of `nurse` and `ins`. It is intended for the confidential communication of the insurance number. The group `hosp` includes both nurses and doctors. The group `pub` includes all peers, it is the public group. Each peer possesses a pair of public\private keys $K_{u,p}$ and $K_{v,p}$ where $p \in \{d, n, i, r\}$. Each group also possesses a shared key S_g where $g \in G$.

6.1. Specifying the Lifecycle

Figure 6 shows two partial lifecycle automata. We show the specification for the groups `ins` and `nurse`. For simplicity we only show the accepting states, all other transitions lead to q^{fail}. We write $\texttt{field}(\texttt{write}, \texttt{v})$ (resp. $\texttt{field}(\texttt{update}, \texttt{v})$ and $\texttt{field}(\texttt{approve})$) to indicate that value v is written in field $field$ (resp, v overwrites the existing value, and the current value is approved). A border action `action` expected from two groups $\texttt{g}_1$ and $\texttt{g}_2$ is denoted by $\underset{g_1,g_2}{\texttt{action}}$. For example, the action $\texttt{num}(\texttt{write}, \texttt{n})$ in $\mathcal{A}_{\text{ins}}$ is a border action associated with the group `nurse` ($\mathcal{S}_{\text{ins}}(\texttt{num}(\texttt{write}, \texttt{n})) = \{\texttt{nurse}\}$). This indicates the filling of the social security number must be made by a peer in the group `nurse`. Furthermore, we can consider a simple integrity constraint governing the format of the insurance number.The insurance number in this instance is text (since it can contain letters). The length of the insurance number is 9 characters. Thus, all actions that manipulate the field must do so following the constraints.

Manipulating the Document. An empty document is created, the first action is performed by the nurse. The nurse fills in the patient name by issuing the action $a_0 = \texttt{name}(\texttt{write}, \texttt{v}_0)$ which writes v_0 in the *name* field. Since the

PA	p	a	g	n	d	i	r
pa_0	n	$\texttt{name}(\texttt{write}, \texttt{v}_0)$	pub	×	×	×	×
pa_1	n	$\texttt{info}(\texttt{write}, \texttt{v}_1)$	hosp	×	×	-	-
pa_2	n	$\texttt{num}(\texttt{write}, \texttt{n})$	nurse/ins	×	-	×	-

Table 2: Resulting Sequences for Peers

patient name is public information (for all peers), the nurse signs it with $S_{\texttt{pub}}$. The initial peer action is then $pa_0^* = \langle E[a_0, S_{\texttt{pub}}], n, \texttt{pub}, h_0 \rangle$. The digest h_0 is computed as follows: $h_0 = E[\hbar(0 \cdot pa_0^* \cdot \texttt{pub}), K_{v,n}]$. The new encrypted action pa_0^* and group information is added to the empty message, appended to digest 0, hashed, and signed by the nurse n using their private key. Since this is the first element in the sequence, the prior message had an empty sequence with digest 0. After entering the name, the nurse enters sensitive patient information using $a_1 = \texttt{info}(\texttt{write}, \texttt{v}_1)$. Since this information is confidential and intended for the hospital only, the nurse signs it with $S_{\texttt{hosp}}$. The second peer-action is then $pa_1^* = \langle E[a_1, S_{\texttt{hosp}}], n, \texttt{hosp}, h_1 \rangle$, with $h_1 = E[\hbar(h_0 \cdot pa_0^* \cdot \texttt{hosp}), K_{v,n}]$. The third peer-action concerns the insurance number and is only shared confidentially between the nurse and the insurance company. The action is $a_2 = \texttt{num}(\texttt{write}, \texttt{n})$, and it is encrypted with $S_{\texttt{nurse/ins}}$ to form pa_2^* with digest h_2, The resulting sequence is $\overline{s}^* = (pa_0^*, pa_1^*, pa_2^*)$.

6.2. Verifying the Sequence

We now consider checking and decrypting the sequence $\overline{s}^*$ from the insurance perspective (peer i). We start from the last peer-action $pa_2 = \langle a_2^*, n, \texttt{nurse/ins}, h_2 \rangle$. We first check the digest of the sequence by testing $\nabla^{-1}(\overline{s}^*, h_2)$. We verify the signature by checking $D[h_2, K_{n,u}] = \hbar(h_1 \cdot a_2^* \cdot \texttt{nurse/ins})$. We continue checking the digest by testing $D[h_1, K_{n,u}] = \hbar(h_0 \cdot a_1^* \cdot \texttt{hosp})$ and so forth. An important thing to note here, is that, while a_1^* is not known to the insurance group at all (as it consists of the private profile of the client), they can still verify the integrity of the peer-action by verifying that the nurse (n) has done it on behalf of the group hosp. However it is impossible to decrypt a_1^* as the insurance company does not have access to $S_{\texttt{hosp}}$. Decrypting the sequence for the insurance is performed using $\text{SD}(\overline{s}^*, i)$. The insurance peer has access to the following group keys: $S_{\texttt{pub}}$, $S_{\texttt{ins}}$, and $S_{\texttt{nurse/ins}}$, it is therefore only capable of decrypting a_0 and a_2. Thus the sequence for insurance is $s_i = (pa_0, pa_2)$. The obtained sequences for all relevant peers are shown in Table 2. The first 4 columns detail parts of the peer-action tuple; they list respectively, the peer-action identifier, the peer that has taken the action, the action itself, the group that encrypted the action. The last 4 columns represent each peer. For each action, we indicate whether the peer-action is present (×) or absent (-) in the sequence for a given peer.

6.3. Verifying the Lifecycle

Once the sequences are decrypted, it is possible to verify the lifecycle by running the peer-action sequence on all automata related to the groups the peer

belongs to. In the case of insurance (i) with sequence $s_i = (pa_0, pa_2)$, we verify it on $\mathcal{A}_{\text{ins}}$ (shown in Figure 6) starting from q_{00}. The first action is $\texttt{name}(\texttt{write}, \texttt{v}_0)$. This action however is not in the alphabet Σ_{ins}, we ignore it and remain in q_{00}. Since q_{00} is not a rejecting state we continue with the second peer-action in the sequence. The action $\texttt{num}(\texttt{write}, \texttt{n})$ (a_2) is the second action, we have indeed a transition to q_{01}. In addition, since a_2 is a border action, we must also check $\mathcal{S}_i(a_2)$. The peer performing the action is the nurse (n). The nurse n belongs to the group $\texttt{nurse} \in \mathcal{S}_i(a_2)$. Therefore, the transition leads to q_{01} which is not a rejecting state, indicating the document is in compliance with the lifecycle. The document's integrity can then be checked to determine whether integrity constraints have been violated (as explained in Section 4.3.2). The rest of $\mathcal{A}_i$ enforces the scenario of drug prescription. An action `prescribe(d)` must be made by a peer in the group `doctor`, followed by a local action of `approve(d)` to approve the prescription. A requirement is added, stating that a prescription will be filled by a pharmacist before another prescription is made. Any sequence not respecting the ordering in the automaton is rejected and therefore incorrect documents can be determined.

7. Implementation and Discussion

To illustrate the concepts introduced in this paper, we implemented a software library, called *Artichoke-X*, that can manipulate and inject peer-action sequences into documents of various types. In this section, we discuss this implementation and report on an experimental evaluation of the use of peer-action sequences in PDF metadata.

7.1. The Artichoke-X Library

Our implementation is composed of two distinct tools. The first is Artichoke-X, a Java library that is freely available under an open source license[7]. Artichoke-X uses the built-in cryptographic functions provided by Java (such as RSA and MD5); for the manipulation of file metadata, it relies on Apache Tika[8], a versatile library that can read and write metadata for more than a thousand file formats. These include commonly used document types such as Microsoft Office documents, PDF files, image and audio files, and even source code.

7.1.1. Library Usage

Using the library is done by programmatically manipulating high-level objects such as peers, actions and groups. The first step is to create (or retrieve) instances of these objects for a specific scenario. In the code snippet below, we create one peer and one group, and associate a pair of RSA public/private keys to each. An instance of an `Action` object is also created. In this case, an action is simply represented by a character string.

[7] `https://github.com/liflab/artichoke-x`
[8] `https://tika.apache.org`

```
KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
Peer alice = new Peer("Alice", Cipher.getInstance("RSA"));
alice.setKeyPair(generator.generateKeyPair());
Group g1 = new Group("Group␣1", Cipher.getInstance("RSA"));
g1.setKeyPair(generator.generateKeyPair());
Action a = new Action("a");
```

The next step is to create an empty peer-action sequence, and to start appending actions to it. This is done by creating a `HistoryManager`, which is in charge of manipulating a `History` object. The manager is first populated with the set of possible peers, groups and actions it can encounter. In the snippet below, a new manager is such created and instructed to use MD5 as its hash function. An action "a", made by by Alice on behalf of group G1, is then appended to the sequence.

```
HistoryManager manager = new HistoryManager(
  MessageDigest.getInstance("MD5"));
manager.add(alice, bob, ...);
manager.add(g1, g2, ...);
manager.add(a, b, ...);
History h = new History();
manager.appendAction(h, alice, a, g1);
```

The history manager can also be instructed to verify an existing peer-action sequence:

```
try {
  manager.isHistoryValid(h);
}
catch (InvalidHistoryException e) {
  // Sequence is invalid
}
```

The call to `isHistoryValid` throws an exception detailing the reason for the violation, and in particular indicating at what position the offending element of the peer-action sequence was found.

The last step of this process is to evaluate a lifecycle policy on a peer-action sequence. This is done by implementing the `Policy` interface. This interface defines two methods; the first, `evaluate`, takes as its input a single element of the history (consisting of a peer, and action and a group). This method is intended to be called successively on every element of a peer-action sequence, and to throw an exception whenever the policy is considered to be violated. The second method, `reset`, simply indicates that the evaluation process is to be started over from the beginning. For example, the following snippet is a policy that checks that Alice cannot execute the action "a" more than once in a document:

```
class AlicePolicy implements Policy {

  boolean seen_a;
```

```
  public void evaluate(Peer p, Action a, Group g)
  throws PolicyViolationException {
    if (p.getName() == "Alice" && a.getName() == "a")
      if (seen_a)
        throw new PolicyViolationException(
          "Alice␣has␣executed␣'a'␣twice");
      seen_a = true;
  }

  public void reset() {
    seen_a = false;
  }
}
```

Given a `Policy` object, the history manager can be told to evaluate it on a given peer-action sequence:

```
Policy alice_pol = new AlicePolicy();
try {
  manager.evaluate(h, alice_pol);
}
catch (PolicyViolationException e) {
  // Policy is violated
}
```

As one can see, the Artichoke-X library is very flexible in many respects. First, it allows the use of arbitrary functions for hashing and public-key encryption. Second, actions can be anything, as long as they can be serialized into a character string. Third, the expression of a lifecycle policy is not restricted to any formal language (such as finite-state machines, temporal logic or BEDL): the latter example has shown how the implementation of the `Policy` object can include arbitrary Java code, and hence subsumes any of these formalisms. However, should users wish to express policies in a higher-level notation, there exist multiple software libraries (such as SealTest [51]) that allow one to write specifications using UML statecharts or Linear Temporal Logic, and wrap them into Artichoke's `Policy` objects.

7.1.2. Command-Line Front-End

We then developed *Artichoke-PDF*, a command-line tool for the specific scenario where an artifact is a dynamic, fillable PDF form. In this context, the various fields of the form constitute the document's data, which can be filled and modified by various peers. A special, hidden form field is included to the document, which is intended to contain the peer-action sequence reflecting the document's modification history.[9]

[9]Note that this field is only made invisible for the sake of readability; its hidden nature has nothing to do with protecting it from tampering.

Artichoke-PDF uses LaTeX to generate forms with various input and an empty peer-action sequence. It also uses pdftk[10] to extract and manipulate form data in the background. Although Artichoke-PDF is intended as a proof-of-concept implementation with minimal user-friendliness, it is fully functional and its source code is public available under the GNU GPL.[11] The current implementation supports a slightly simplified version of peer-action sequences, where a single group exists, but peers in the group each have their own public/private key pair to stamp their actions.

Currently, Artichoke-PDF supports the three main operations on a document, namely filling, examining and checking the peer-action sequence of a form.

A first operation is to fill a form, which consists in writing (or overwriting) one or more form fields with specified values. In our context, filling a form also involves updating the peer-action sequence contained in that form to include the modification action and peer information related to that action.

This can be done through the command line as follows. For example, if Alice wants to write "foo" to field `F1` of `Form.pdf`, the command is:

```
$ artichoke Form.pdf fill \
  -k private_key_Alice.pem \
  -p Alice \
  -o Form-filled.pdf \
  F1 foo
```

Here, command line argument `-p` specifies the name of the peer, `-k` specifies the private key to use when computing the digest, and `-o` gives the name modified PDF file.

A second operation is to examine the contents of a form; this is performed by issuing a command such as:

```
$ artichoke Form.pdf dump
```

This will print the current value of all the form's fields, and display a summary of the peer-action sequence contained in the document, which will look like this:

```
Form fields
-----------
F1:     baz
F2:     bar

Peer-action sequence
--------------------
Alice   W|F1|foo       Rm/MRSzK...oYpROg0=
Bob     W|F2|bar       kEvrkC+e...bX4NO1w=
Carl    W|F1|baz       F3UYg+n1.../YPs3/k=
```

[10] `http://pdftk.org`
[11] `https://github.com/liflab/artichoke-pdf`

The peer-action sequence shows that Alice first wrote “foo” to field F1, then Bob wrote "bar" to field F2, then Carl overwrote F1 with “baz”. The rightmost column is a shortened version of the digest string for each event.

The last operation that can be done with Artichoke is to validate the contents and history of a form. ; this is done as follows:

```
$ artichoke Form.pdf check key1 [key2 [...]]
```

This will check the peer-action sequence in `filename`, using public key filenames `key1`, `key2`, etc. if necessary. This list of local filenames effectively acts as a primitive form of “keyring”. Obviously, a more mature version of the tool could replace these files by the user's local machine keyring, or even query remote servers storing X.509 public key certificates.

The check operation will perform the three verification steps mentioned earlier, that is: 1. Making sure the digest of each event in the peer-action sequence is consistent with the action and peer name specified 2. Making sure the values of each field in the form match the result of applying the sequence of actions to an empty document 3. Making sure the sequence of actions follows the policy.

The policy is currently specified through user-defined PHP code, by implementing a special function called `check_policy` that receives as its input the peer-action sequence of the current document. Hence, as for the Java library, the enforcement of a policy is not tied to any particular specification language, provided it can be expressed in terms of the contents of peer-action sequence only.

7.2. Resource Consumption

We proceeded to perform tests intended to measure the computational resources required in a typical use-case scenario. In particular, we want to determine whether the repeated application of encryption and hashing induces a reasonable cost, in terms of both time and space, as the history of a document lengthens over time.

As is now customary for projects made at LIF[12], the experiments were implemented using the LabPal testing framework [52]. The principle behind LabPal is that all the necessary code, libraries and input data should be bundled within a single self-contained executable file, such that anyone can download and easily reproduce someone else's experiments.[13]

As the overhead of reading and writing to a file is irrelevant to our study, the experiments were performed by directly manipulating the sequence with the Artichoke-X library. The experiments were done on a mid-range Lenovo A20 computer with 4 GB of RAM, running under Ubuntu 16.04.1.

[12]Laboratoire d'informatique formelle, the research lab where some of the authors work.

[13]The lab for this paper can be downloaded from `https://datahub.io/dataset/artichoke-x-lab`.

We first generated an empty peer-action sequence, and repeatedly appended dummy actions on behalf of a random user and group. This had the effect of creating a peer-action sequence of increasing length.

The first factor we measured is the running time for appending a new action to an existing sequence. This is shown in Figure 7a. One can see that the running time increases linearly with the number of write operations. We can deduce from this graph that it takes approximately 2.5 milliseconds to perform a single append operation.

The second factor we measured is the time required to simply *check* an existing sequence without modifying it; this is shown in Figure 7b. Again, this verification time is linear in the length of the sequence (as expected), with a verification time per element averaging 0.15 millisecond. One can see that read/decrypt operations are much quicker to perform than write/encrypt ones.

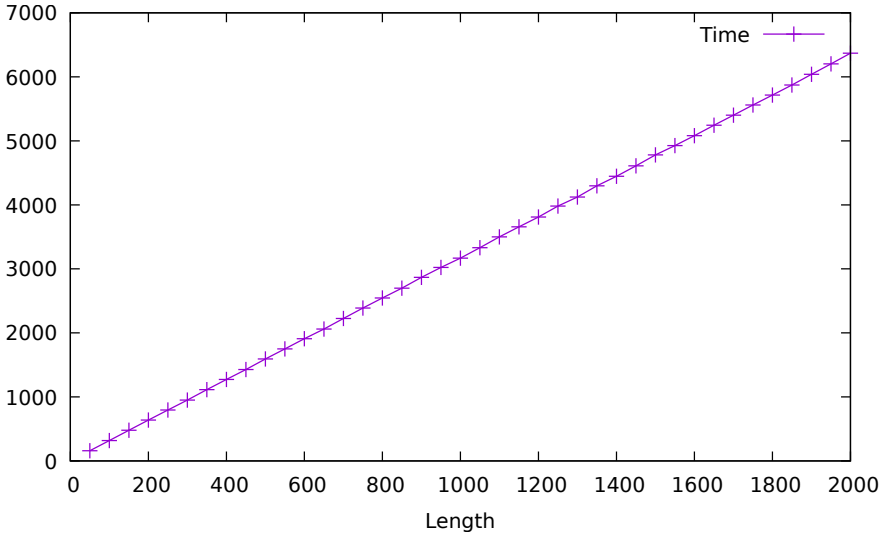
However, Section 5.10 has introduced the concept of *stateful peer*, where a peer reserves a small amount of persistent space to save the last element of the peer-action sequence it has processed, and its position i inside the sequence. When receiving a peer-action sequence extending a previous one, such a peer can simply check that the i-th element of the sequence is identical to the one kept in memory, and then verify only the part of the sequence that has been appended after this position. A stateless peer, on the contrary, has no memory of the sequence; when handed a peer-action sequence, it must re-validate it from the start.

Therefore, another experiment we performed consists of comparing the total processing time required to verify a peer-action sequence of a given length, between a stateful and a "stateless" peer. The results are shown in Figure 8. As expected, the use of a stateful peer dramatically improves the time required to process a sequence. While it takes a total of 5 seconds to incrementally check a sequence of length 100 with a stateful peer, the same operations on a stateless peer require about 4 minutes. The gap between the two methods even widens with the length of the sequence.

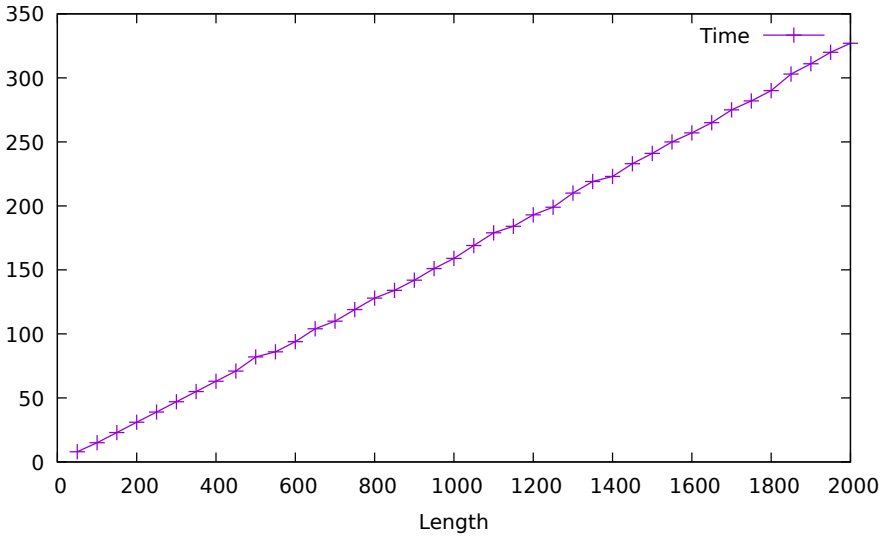
The final factor we measured is the size of the sequence for increasing lengths of a peer-action sequence; this is shown in Figure 9. As expected, the size of the sequence grows linearly with the number of operations applied to the document, indicating that each element of the sequence requires constant space. In the current implementation of Artichoke-X, this space amounts to roughly 450 bytes per action. Note however that the default form of serialization in the library is particularly inefficient, as it uses two passes of Base-64 encoding to convert binary hashes into character strings. To reduce space consumption, more compact forms of encoding could easily be used (such as representing hashes as hexadecimal strings). Nevertheless, even in its current form, one can argue that encoding peer-action sequences into a document incur a reasonable space overhead, with one megabyte containing more than 2,000 distinct modifications.

7.3. Discussion

Overall, the positive results obtained with the current implementation of Artichoke illustrate the potential of peer-action sequences to effectively encode



(a) Writing to the document



(b) Checking the document

Figure 7: Running time of Artichoke on PDF documents with a peer-action sequence of increasing length: (a) to write the to document; (b) to check a document.

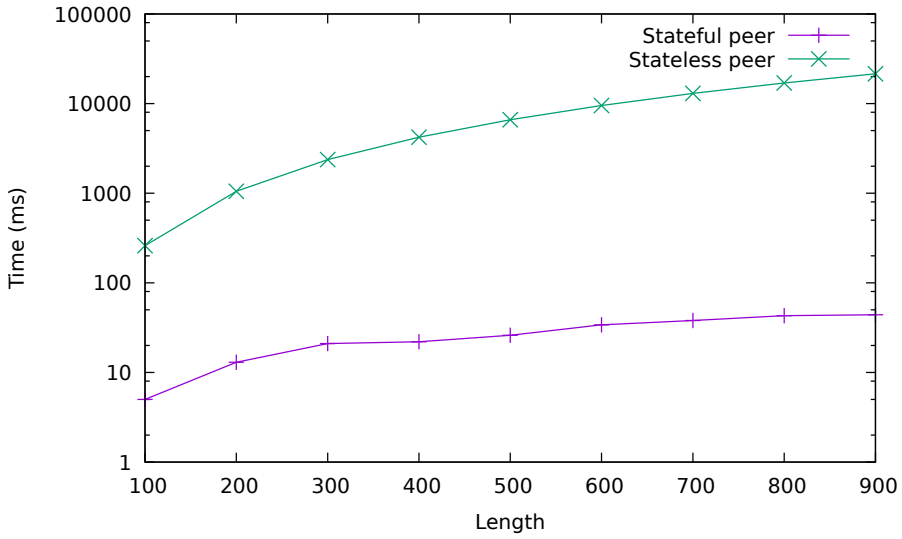


Figure 8: Comparison of processing time between a stateful and a stateless peer, to verify the same peer-action sequence. Notice the use of a logarithmic scale on the y axis.

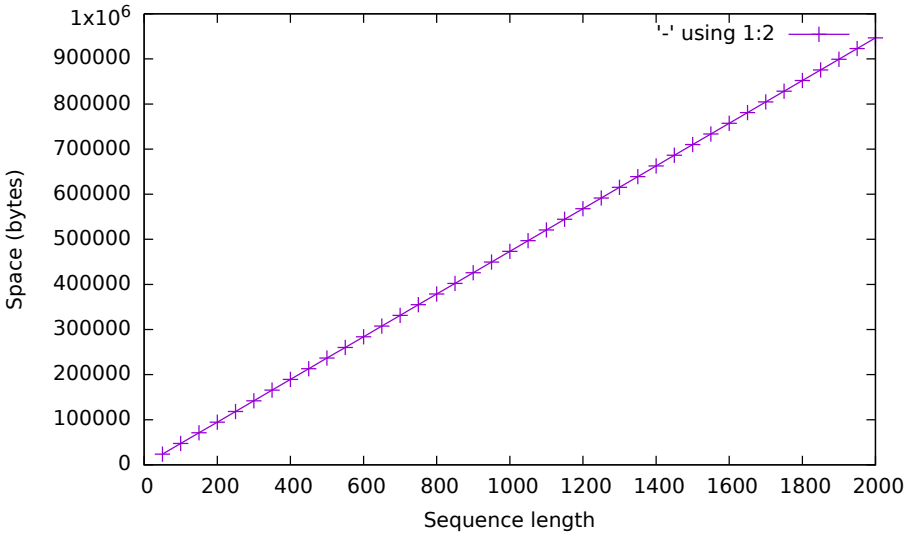


Figure 9: File size of a PDF document with a peer-action sequence of increasing length.

a document's history so that lifecycle constraints can be verified on it at any moment. We mention in the following a few discussion points regarding the current system.

7.3.1. Space requirements

In addition to the document's contents, storage space is required to hold the peer-action sequence, whose size is proportional to the length of the history. Note that in the general setting, this sequence cannot simply be trimmed of its first events after "long enough", as a peer could use this facility to cover up a fraudulent manipulation of the document. The question remains open whether *stateless* peers can be given any freedom in erasing prefixes of the history without the possibility of misuse. (The case of stateful peers was discussed in Section 5.10.)

7.3.2. Enforcement

In the proposed system, the enforcement of lifecycle constraints is indirect. Any peer can tamper with the contents of the document, with its history, or perform modifications that violate the lifecycle requirements. Likewise, any peer can choose to accept such a tampered document, modify it and pass it on to other peers. However, our approach makes sure that anyone with knowledge of the peers' public keys (including peers external to the exchange) can check at any time whether such misuses occurred, as well as pinpoint what peers have been faulty or complacent.

7.3.3. Duplication

In some situations, the document can be duplicated. Therefore, a peer can receive a document, modify it in two different ways, and pass it on to two different peers. Our proposed approach will still ensure that each copy will follow a compliant lifecycle, but the uniqueness of each document cannot be ensured. However, since our approach allows the specification of a lifecycle for a document, conditions can be added to this lifecycle so that uniqueness is guaranteed. One simple (and relatively restrictive) condition could be that at any point, the possible sender for the next action is always unique (and would henceforth detect if the same document is sent twice). Determining conditions for uniqueness is outside the scope of this work.

7.3.4. Applicability

The remaining, and perhaps most important question, is the issue of the applicability of this technique in real-world scenarios. Although no full-scale experiment of an implementation in an actual situation has yet been performed, the findings detailed in Section 7.2 allow us to draw a few conclusions.

First off, our proposed technique is agnostic with respect to the actual content of actions (which are simply interpreted as streams of bytes) and to the actual policy that is being enforced (which is taken as a "black box" that is handed the peer-action sequence once it has been checked for validity). Therefore, the fact

that actions represent anything meaningful has no bearing on the computation times reported earlier. From these figures, a peer with modest computing power, being handed a peer-action sequence to be extended by one more action, would require 350 ms to first check that the trace is valid, and 2.5 ms to append a new action to that trace. This is in the case where the sequence already contains 2,000 actions; a shorter sequence would yield an even shorter checking time. Therefore, a safe assumption is that a single read-check-append cycle can be done in under half a second, and possibly much faster. This is in the situation where none of the peers persist any information but the keys that need to be managed.

These verification times are reasonable for a large set of the situations described in Section 3. Clearly, in the medical form example, adding a half-second delay when opening a PDF document is probably perfectly acceptable; moreover, the validation of the sequence could even be done in the background, making the document immediately available while its correctness is being checked in parallel. In the metro card example, a half-second checking time at the turnstile is also close to acceptable, given the time it takes for a passenger to simply cross the apparatus.

The amount of information that needs to be stored in the peer-action sequence is also reasonable. Our fairly inefficient scheme of 450 bytes per action, which is probably well enough for files such as PDFs, could easily be reduced in half by using more compact representations of hashes and strings. This entails that 8 kb (the size of a large MIFARE card) could store roughly 30 actions —arguably enough, in the metro card example, to follow a passenger's comings and goings for a whole day, assuming that a card's history could be safely wiped every day.

8. Conclusions and Future Work

In this paper, we have shown how the lifecycle of an artifact can be effectively stored within the document itself, using the concept of peer-action sequences. Moreover, this sequence can be protected from tampering through an appropriate use of public-key encryption and hashing. This provides at the same time a mechanism for enforcing different read-write access permissions to various parts of the document, depending on the *group* a peer belongs to. Experiments have shown that manipulating these sequences does not impose an undue burden in terms of computing resources, and that the space required to store a sequence within a document increases linearly with the number of modifications made to it. Combined, these observations tend to indicate that the application of peer-action sequences in real-world situations is feasible: a proof-of-concept implementation of such a system has even been devised in the case of PDF forms.

The main advantage of peer-action sequences, over existing lifecycle compliance approaches, is the fact that compliance can be checked on-the-fly and at any moment on a document that can be freely exchanged between peers. Peers do not need to be statically verified prior to any interaction, and the document is not required to be accessed from a single point in order to enforce

compliance. This presents the potential of greatly simplifying the implementation of artifact-centric workflows, by dropping many assumptions that must be fulfilled by current systems. Taken to its extreme, lifecycle policies can even be verified without resorting to any workflow management system at all: as long as documents are properly stamped by everybody, the precise way they are exchanged (e-mail, file copying, etc.) is irrelevant. Technically speaking, the next step of this work will be to imagine lifecycle policies for types of documents not traditionally considered by the business process community —such as restrictions on the way image files can be manipulated. In the case of forms, the filling, stamping and compliance checking of PDF files with respect to a peer-action sequence could be implemented directly into the graphical user interface of a PDF reader, and become a seamless process that could be executed by a user in a single button click.

On the formal side, a number of possible extensions and open questions also arise. For example, could we enforce proper usage by rendering the document unreadable if improperly modified? This way a peer would not even need to replay the history: simply trying to read the document would reveal a problem. The enforcement of constraints across multiple documents in the same lifecycle is also an open issue; the use of synchronization signals between peers, borrowed from decentralized runtime monitoring, could prove a promising solution. Finally, the question of uniqueness of documents also needs to be studied. In its current incarnation, the proposed system allows artifacts to be duplicated, yet enforces that all copies must follow a valid lifecycle.

References

[1] S. Hallé, R. Khoury, A. El-Hokayem, Y. Falcone, Decentralized enforcement of artifact lifecycles, in: F. Matthes, J. Mendling, S. Rinderle-Ma (Eds.), 20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, Vienna, Austria, September 5-9, 2016, IEEE Computer Society, 2016, pp. 1–10. doi:10.1109/EDOC.2016.7579380.
URL https://doi.org/10.1109/EDOC.2016.7579380

[2] N. Bielova, F. Massacci, A. Micheletti, Towards practical enforcement theories, in: Proceedings of The 14th Nordic Conference on Secure IT Systems, Vol. 5838 of Lecture Notes in Computer Science, Springer-Verlag Heidelberg, 2009, pp. 239–254.

[3] N. Bielova, F. Massacci, Predictability of enforcement, in: Proceedings of the International Symposium on Engineering Secure Software and Systems 2011, Vol. 6542, Springer, 2011, pp. 73–86.

[4] R. R. Mukkamala, T. T. Hildebrandt, J. B. Tøth, The resultmaker online consultant: From declarative workflow management in practice to LTL, in: M. van Sinderen, J. P. A. Almeida, L. F. Pires, M. Steen (Eds.), EDOCW, IEEE Computer Society, 2008, pp. 135–142. doi:10.1109/EDOCW.2008.57.
URL http://dx.doi.org/10.1109/EDOCW.2008.57

[5] P. W. L. Fong, Access control by tracking shallow execution history, in: S&P, 2004, pp. 43–55. doi:10.1109/SECPRI.2004.1301314.
URL http://dx.doi.org/10.1109/SECPRI.2004.1301314

[6] W. Boebert, R. Kain, A practical alternative to hierarchical integrity policies, in: S&P, 1985.

[7] D. F. C. Brewer, M. J. Nash, The Chinese wall security policy, in: S&P, IEEE Computer Society, 1989, pp. 206–214. doi:10.1109/SECPRI.1989.36295.
URL http://dx.doi.org/10.1109/SECPRI.1989.36295

[8] A. E. K. Sobel, J. Alves-Foss, A trace-based model of the Chinese wall security policy, in: Proc. of the 22nd National Information Systems Security Conference, 1999.

[9] K. J. Biba, Integrity considerations for secure computer systems, Tech. rep., MITRE Corporation (1977).

[10] H. D. Johansen, E. Birrell, R. Van Renesse, F. B. Schneider, M. Stenhaug, D. Johansen, Enforcing privacy policies with meta-code, in: Proceedings of the 6th Asia-Pacific Workshop on Systems, ACM, 2015, p. 16.

[11] International Council of E-Commerce Consultants, Computer Forensics: Investigating Network Intrusions and Cyber Crime, Cengage Learning, 2009.

[12] A. Nigam, N. Caswell, Business artifacts: An approach to operational specification, IBM Syst. J. 42 (3) (2003) 428–445.

[13] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, J. Su, Towards formal analysis of artifact-centric business process models, in: G. Alonso, P. Dadam, M. Rosemann (Eds.), BPM, Vol. 4714 of Lecture Notes in Computer Science, Springer, 2007, pp. 288–304. doi:10.1007/978-3-540-75183-0_21.
URL http://dx.doi.org/10.1007/978-3-540-75183-0_21

[14] S. Kumaran, R. Liu, F. Y. Wu, On the duality of information-centric and activity-centric models of business processes, in: Z. Bellahsene, M. Léonard (Eds.), CAiSE, Vol. 5074 of Lecture Notes in Computer Science, Springer, 2008, pp. 32–47. doi:10.1007/978-3-540-69534-9_3.
URL http://dx.doi.org/10.1007/978-3-540-69534-9_3

[15] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. H. Linehan, S. Maradugu, A. Nigam, P. N. Sukaviriya, R. Vaculín, Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events, in: D. M. Eyers, O. Etzion, A. Gal, S. B. Zdonik, P. Vincent (Eds.), DEBS, ACM, 2011, pp. 51–62. doi:10.1145/2002259.2002270.
URL http://doi.acm.org/10.1145/2002259.2002270

[16] R. Vaculín, R. Hull, T. Heath, C. Cochran, A. Nigam, P. Sukaviriya, Declarative business artifact centric modeling of decision and knowledge intensive business processes, in: EDOC, IEEE Computer Society, 2011, pp. 151–160. doi:10.1109/EDOC.2011.36.
URL http://dx.doi.org/10.1109/EDOC.2011.36

[17] P. Nandi, D. Koenig, S. Moser, R. Hull, V. Klicnik, S. Claussen, M. Kloppmann, J. Vergo, Data4BPM, part 1: Introducing business entities and the business entity definition language (BEDL) (April 2010).

[18] A. Meyer, L. Pufahl, D. Fahland, M. Weske, Modeling and enacting complex data dependencies in business processes, in: F. Daniel, J. Wang, B. Weber (Eds.), BPM, Vol. 8094 of Lecture Notes in Computer Science, Springer, 2013, pp. 171–186.

[19] V. Künzle, M. Reichert, Philharmonicflows: towards a framework for object-aware process management, J. of Software Maintenance 23 (4) (2011) 205–244.

[20] B. B. Hariri, D. Calvanese, G. De Giacomo, R. De Masellis, P. Felli, Foundations of relational artifacts verification, in: S. Rinderle-Ma, F. Toumani, K. Wolf (Eds.), BPM, Vol. 6896 of Lecture Notes in Computer Science, Springer, 2011, pp. 379–395. doi:10.1007/978-3-642-23059-2_28.
URL http://dx.doi.org/10.1007/978-3-642-23059-2_28

[21] S. Pearson, M. C. Mont, Sticky policies: An approach for managing privacy across multiple parties, Computer 44 (9) (2011) 60–68.
URL https://documents.epfl.ch/users/a/ay/ayday/www/mini_project/Sticky%20Policies.pdf

[22] E. Birell, F. B. Schneider, Fine-grained user privacy from avenance tags.
URL http://hdl.handle.net/1813/36285

[23] J. Camenisch, A. Shelat, D. Sommer, S. Fischer-Hübner, M. Hansen, H. Krasemann, G. Lacoste, R. Leenes, J. Tseng, Privacy and identity management for everyone, in: Proceedings of the 2005 Workshop on Digital Identity Management, DIM '05, ACM, New York, NY, USA, 2005, pp. 20–27. doi:10.1145/1102486.1102491.
URL https://dx.doi.org/10.1145/1102486.1102491

[24] Extensible access control markup language (XACML) version 3.0, Standard, OASIS Standard (Jan. 2013).
URL http://docs.oasis-open.org/xacml/3.0/

[25] X. Zhao, J. Su, H. Yang, Z. Qiu, Enforcing constraints on life cycles of business artifacts, in: W. Chin, S. Qin (Eds.), TASE, IEEE Computer Society, 2009, pp. 111–118. doi:10.1109/TASE.2009.46.
URL http://dx.doi.org/10.1109/TASE.2009.46

[26] A. A. Ataullah, F. W. Tompa, Business policy modeling and enforcement in databases, PVLDB 4 (11) (2011) 921–931.
URL http://www.vldb.org/pvldb/vol4/p921-ataullah.pdf

[27] D. Calvanese, G. De Giacomo, R. Hull, J. Su, Artifact-centric workflow dominance, in: L. Baresi, C. Chi, J. Suzuki (Eds.), ICSOC-ServiceWave, Vol. 5900 of Lecture Notes in Computer Science, 2009, pp. 130–143. doi:10.1007/978-3-642-10383-4_9.
URL http://dx.doi.org/10.1007/978-3-642-10383-4_9

[28] C. E. Gerede, J. Su, Specification and verification of artifact behaviors in business process models, in: B. J. Krämer, K. Lin, P. Narasimhan (Eds.), ICSOC, Vol. 4749 of Lecture Notes in Computer Science, Springer, 2007, pp. 181–192. doi:10.1007/978-3-540-74974-5_15.
URL http://dx.doi.org/10.1007/978-3-540-74974-5_15

[29] S. Hallé, R. Villemaire, O. Cherkaoui, Specifying and validating data-aware temporal web service properties, IEEE Trans. Software Eng. 35 (5) (2009) 669–683. doi:10.1109/TSE.2009.29.
URL http://dx.doi.org/10.1109/TSE.2009.29

[30] P. Gonzalez, A. Griesmayer, A. Lomuscio, Verifying gsm-based business artifacts, in: C. A. Goble, P. P. Chen, J. Zhang (Eds.), 2012 IEEE 19th International Conference on Web Services, Honolulu, HI, USA, June 24-29, 2012, IEEE Computer Society, 2012, pp. 25–32. doi:10.1109/ICWS.2012.31.
URL http://dx.doi.org/10.1109/ICWS.2012.31

[31] D. Calvanese, M. Montali, M. Estañol, E. Teniente, Verifiable UML artifact-centric business process models, in: J. Li, X. S. Wang, M. N. Garofalakis, I. Soboroff, T. Suel, M. Wang (Eds.), Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014, ACM, 2014, pp. 1289–1298. doi:10.1145/2661829.2662050.
URL http://doi.acm.org/10.1145/2661829.2662050

[32] S. Hallé, R. Villemaire, Runtime enforcement of web service message contracts with data, IEEE Trans. Services Computing 5 (2) (2012) 192–206. doi:10.1109/TSC.2011.10.
URL http://dx.doi.org/10.1109/TSC.2011.10

[33] Y. Falcone, L. Mounier, J. Fernandez, J. Richier, Runtime enforcement monitors: composition, synthesis, and enforcement abilities, Formal Methods in System Design 38 (3) (2011) 223–262. doi:10.1007/s10703-011-0114-4.

[34] Y. Falcone, T. Jéron, H. Marchand, S. Pinisetty, Runtime enforcement of regular timed properties by suppressing and delaying events, Systems & Control Letters 123 (2016) 2–41. doi:10.1016/j.scico.2016.02.008.
URL http://dx.doi.org/10.1016/j.scico.2016.02.008

[35] Y. Falcone, You should better enforce than verify, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, Vol. 6418 of Lecture Notes in Computer Science, Springer, 2010, pp. 89–105. doi:10.1007/978-3-642-16612-9_9.
URL http://dx.doi.org/10.1007/978-3-642-16612-9_9

[36] K. Ouafi, S. Vaudenay, Pathchecker: an RFID application for tracing products in supply-chains, in: RFIDSec, 2009, pp. 1–14.

[37] A. K. Bauer, Y. Falcone, Decentralised LTL monitoring, in: D. Giannakopoulou, D. Méry (Eds.), FM, Vol. 7436 of Lecture Notes in Computer Science, Springer, 2012, pp. 85–100. doi:10.1007/978-3-642-32759-9_10.
URL http://dx.doi.org/10.1007/978-3-642-32759-9_10

[38] C. Colombo, Y. Falcone, Organising LTL monitors over distributed systems with a global clock, in: B. Bonakdarpour, S. A. Smolka (Eds.), RV, Vol. 8734 of Lecture Notes in Computer Science, Springer, 2014, pp. 140–155. doi:10.1007/978-3-319-11164-3_12.
URL http://dx.doi.org/10.1007/978-3-319-11164-3_12

[39] A. Bauer, Y. Falcone, Decentralised LTL monitoring, Formal Methods in System Design 48 (1-2) (2016) 46–93. doi:10.1007/s10703-016-0253-8.
URL http://dx.doi.org/10.1007/s10703-016-0253-8

[40] C. Colombo, Y. Falcone, Organising LTL monitors over distributed systems with a global clock, Formal Methods in System Design 49 (1-2) (2016) 109–158. doi:10.1007/s10703-016-0251-x.
URL http://dx.doi.org/10.1007/s10703-016-0251-x

[41] Y. Falcone, T. Cornebize, J. Fernandez, Efficient and generalized decentralized monitoring of regular languages, in: E. Ábrahám, C. Palamidessi (Eds.), FORTE, Vol. 8461 of Lecture Notes in Computer Science, Springer, 2014, pp. 66–83. doi:10.1007/978-3-662-43613-4_5.
URL http://dx.doi.org/10.1007/978-3-662-43613-4_5

[42] S. Hallé, Cooperative runtime monitoring, Enterprise IS 7 (4) (2013) 395–423. doi:10.1080/17517575.2012.721118.
URL http://dx.doi.org/10.1080/17517575.2012.721118

[43] G. Spyra, W. J. Buchanan, E. Ekonomou, Sticky policy enabled authenticated ooxml, in: 2016 SAI Computing Conference (SAI), IEEE. doi:10.1109/SAI.2016.7556117.
URL https://dx.doi.org/10.1109/SAI.2016.7556117

[44] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008).
URL https://bitcoin.org/bitcoin.pdf

[45] K. Okupski, Bitcoin developer reference (2016).
URL https://github.com/minium/Bitcoin-Spec/blob/master/Bitcoin.pdf

[46] D. Ferraiolo, D. Kuhn, Role-based access control, in: S&P, 1992, p. 554–563.

[47] Attribute-based access control, Tech. rep., https://www.jerichosystems.com/technology/glossaryterms/attribute_based_access_control.html (2017).

[48] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web service description language, Tech. rep., https://www.w3.org/TR/wsdl (2001).

[49] T. Wilke, Classifying discrete temporal properties, in: C. Meinel, S. Tison (Eds.), STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings, Vol. 1563 of Lecture Notes in Computer Science, Springer, 1999, pp. 32–46. doi:10.1007/3-540-49116-3_3.
URL http://dx.doi.org/10.1007/3-540-49116-3_3

[50] G. Ateniese, M. Steiner, G. Tsudik, New multiparty authentication services and key agreement protocols, IEEE Journal on Selected Areas in Communications 18 (4) (2000) 628–639. doi:10.1109/49.839937.
URL http://dx.doi.org/10.1109/49.839937

[51] S. Hallé, R. Khoury, SealTest: a simple library for test sequence generation, in: Bultan and Sen [53], pp. 392–395. doi:10.1145/3092703.3098229.
URL http://doi.acm.org/10.1145/3092703.3098229

[52] S. Hallé, LabPal: repeatable computer experiments made easy, in: Bultan and Sen [53], pp. 404–407. doi:10.1145/3092703.3098232.
URL http://doi.acm.org/10.1145/3092703.3098232

[53] T. Bultan, K. Sen (Eds.), Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017, ACM, 2017. doi:10.1145/3092703.
URL http://doi.acm.org/10.1145/3092703